Habilitation Thesis

Symmetric Cryptanalysis Beyond Primitives

Gaëtan Leurent

Defended on January 18, 2024

Reviewers:

Henri Gilbert	*ANSSI*
Tetsu Iwata	*Nagoya University*
Bart Preneel	*KU Leuven*

Jury:

Karthikeyan Bhargavan	*INRIA & Cryspen*
Anne Canteaut	*INRIA*
Orr Dunkelman	*University of Haifa*
Henri Gilbert	*ANSSI*
Tetsu Iwata	*Nagoya University*
Antoine Joux	*CISPA*
Gregor Leander	*Ruhr University Bochum*
Bart Preneel	*KU Leuven*
Vincent Rijmen	*KU Leuven*

À Loreto.

Acknowledgments

First, I must specially thank Anne, María and Jean-Pierre who pushed me to finally write down this manuscript and defend this habilitation. Without their regular nagging, I wouldn't be writing this acknowledgment and you wouldn't be reading it.

I would like to thank all my coauthors, without whom I wouldn't have obtained all the results presented in this thesis. In particular, I would like to thank the PhD students that I have had the pleasure to supervise: Sébastien, Ferdinand, Clara and Augustin. Working with them has been motivating and encouraging.

I am grateful to Pierre-Alain who took me in as a PhD student, and to Alex and François-Xavier who hosted me as a postdoc, for giving me the taste of research in cryptography, a challenging and fun research topic. Since 2013, I have had the pleasure of working with the Secret team at INRIA, which become known as Cosmiq in 2020. I would like to thank all past and present members of the team for welcoming me in the group, for the great interactions we've had, and for the wonderful working environment. The team owes a lot to Anne, who lead the group when it was known as Secret, and to Jean-Pierre as the current leader, but also to all the researchers, post-docs, PhD students and visitors (I will not attempt to list them because I would certainly forget someone). I particularly enjoy the discussions at the coffee break, the daily crosswords and regular occasions to share a drink together. In recent years, the atmosphere at INRIA has become less welcoming to research than it used to be, but the Cosmiq team is still a great place to work, thanks to its members.

Research is a collaborative endeavor, and I'm glad we have a great community in symmetric cryptography, so that it's always fun to meet coauthors and colleagues at conferences, or to spend some time visiting their labs. Within France, we also have regular collaboration with several groups through collaborative projects (in particular the groups in Versailles, Rennes, and Nancy), and this often led to fruitful collaborations; more generally, the C2 community is full of amazing researchers.

Finally, I would like to warmly thank Bart, Henri and Tetsu for their work reviewing my manuscript, and all the members of the jury who make me the honor of being here today: Anne, Antoine, Bart, Gregor, Henri, Karthik, Orr, Tetsu, and Vincent.

Contents

0. Overview of my Results

I study the construction and analysis of symmetric cryptography algorithms. Symmetric primitives are mostly standardized through open international competitions, where their security is assessed by public cryptanalysis. I have contributed to several competitions (SHA-3, CAESAR, NIST Lightweight) by designing new primitives, and analyzing the other candidates. I have also applied cryptanalysis techniques *beyond primitives*, to analyze the security of modes of operations, and to extend cryptanalysis results to practical settings.

I briefly mention my main results below.

Design of primitives. I have contributed to the design of several primitives; most of them have been submitted to standardization processes:

- SIMD, a hash function submitted to the SHA-3 competition [LBF08];
- SPRING, a PRF based on a lattice construction [BBL+15];
- SCREAM an authenticated encryption algorithm based on LS-designs submitted to the CAESAR competition [GLS+14];
- Spook an authenticated encryption algorithm based on LS-designs submitted to the NIST lightweight standardization [BBB+20];
- Saturnin [CDL+20] an authenticated encryption algorithm with post-quantum security submitted to the NIST lightweight standardization [BBB+20]; we later proposed a dedicated mode of operation, with post-quantum security: QCB [BBC+21].

I have also proposed constructions for components of symmetric primitives (used in some of the primitives above):

- the LS-design methodology for block ciphers [GLS+15];
- constructions for lightweight 8-bit S-Boxes based on 4-bit S-Boxes [CDL16];
- constructions of lightweight MDS matrices [DL18].

Cryptanalysis of primitives. I have obtained important cryptanalysis results on primitives, either in the context of standardization processes, or by evaluating the security of widely used primitives. Some of my main results are:

- the first preimage attack against the full MD4 [Leu08a];
- tools for the analysis of ARX primitives [Leu12; Leu13];
- a differential-linear attack against Chaskey [Leu16b]; the attack breaks 7 rounds and has motivated the authors to increase the number of rounds from 8 to 12;
- an analysis of differential and linear cryptanalysis in the quantum setting [KLL+16b];
- a chosen-prefix collision attack against the full SHA-1 [LP19; LP20];
- a new representation of the AES key-schedule [LP21] (*Eurocrypt 2021 best paper award*);

- attacks against the full permutation Gimli [FLN+20; FLN+21] (*Asiacrypt 2020 best paper award*);
- attacks against Simon and Simeck [LPS21];
- an analysis of GEA, the encryption algorithm used in GPRS showing an intentional weakness in GEA-1[BDL+21];
- improvement of boomerang attacks against AES-based primitives [BL23];
- algebraic attacks against arithmetization-oriented primitives [BCD+20; BBL+22; LMP+23]; in particular we solved several challenges from the Ethereum Foundation.[1]

Generic attacks. I also use cryptanalysis techniques more originally, in order to devise generic attacks against modes of operations or ideal constructions. Some of the results obtained are quite surprising:

- generic attacks against hash-based MACs [LPW13; DL14; DL16]; surprisingly, we obtain a key-recovery attack when the hash function uses an internal checksum, showing that the checksum makes the hash function *weaker*;
- a preimage attack against the sum of two hash functions, showing that the sum is often *weaker* than the initial functions [LW15];
- generic attack against several MACs with security beyond the birthday bound [LNS18]; this contributed to closing the gap between proofs and attacks, and we have also invalidated a previous proof;
- a generic attack against the `CTR` mode, showing plaintext recovery with complexity $2^{n/2}$ [LS18];
- forgery attacks with polynomial complexity against MACs in the quantum setting [KLL+16a; BLN+21]; this shows that quantum computing has a strong impact on symmetric cryptography, in addition to asymmetric cryptography.

Practical impact of cryptanalysis. Finally, I'm also interested in the practical security impact of cryptanalysis, and I've demonstrated several attacks against concrete protocols, using weaknesses of cryptographic algorithms. In particular, I am proud that several of my works have contributed to the deprecation of obsolete standards:

- a password-recovery attack against APOP using MD5 collisions [Leu07] (CVE-2007-1558);
- a demonstration of the well-known collision attack with complexity $2^{n/2}$, showing that it is actually practical against block ciphers with 64-bit blocks, such as 3DES (`Sweet32`, CVE-2016-2183, CVE-2016-6329);
- the first chosen-prefix collision for SHA-1, using it for impersonation attack against the *PGP* web-of-trust [LP20] (`Shambles`, CVE-2019-14855);
- transcript-collision attacks against internet protocols `TLS`, `SSH`, and `IPsec` using chosen-prefix collisions for MD5 and SHA-1 [BL16b] (`SLOTH`, CVE-2015-7575 – *NDSS 2016 distinguished paper award*).

Focus of this thesis. In this thesis, I will explain cryptanalysis techniques that don't target directly primitives; instead I will focus on application to modes of operations and ideal constructions, and on leveraging weaknesses of primitives to break concrete protocols.

[1] https://crypto.ethereum.org/bounties/zk-hash

Journals papers

[Leu08b] G. Leurent. “Practical key-recovery attack against APOP, an MD5-based challenge-response authentication”. In: *Int. J. Appl. Cryptogr.* 1.1 (2008), pp. 32–46. DOI: 10.1504/IJACT.2008.017049.

[DL16] I. Dinur and G. Leurent. “Improved Generic Attacks Against Hash-Based MACs and HAIFA”. In: *Algorithmica* (2016), pp. 1–35. ISSN: 1432-0541. DOI: 10.1007/s00453-016-0236-6.

[BDG+20] Z. Bao, I. Dinur, J. Guo, G. Leurent, and L. Wang. “Generic Attacks on Hash Combiners”. In: *Journal of Cryptology* 33.3 (July 2020), pp. 742–823. DOI: 10.1007/s00145-019-09328-w.

[FLN+21] A. Flórez-Gutiérrez, G. Leurent, M. Naya-Plasencia, L. Perrin, A. Schrottenloher, and F. Sibleyras. “Internal Symmetries and Linear Properties: Full-permutation Distinguishers and Improved Collisions on Gimli”. In: *Journal of Cryptology* 34.4 (Oct. 2021), p. 45. DOI: 10.1007/s00145-021-09413-z.

[DEK+22] O. Dunkelman, M. Eichlseder, D. Kales, N. Keller, G. Leurent, and M. Schofnegger. “Practical key recovery attacks on FlexAEAD”. In: *Designs, Codes and Cryptography* 90.4 (Apr. 2022), pp. 983–1007. ISSN: 1573-7586. DOI: 10.1007/s10623-022-01023-5.

Peer-reviewed International Conferences

[CL05] C. Cid and G. Leurent. “An Analysis of the XSL Algorithm”. In: *ASIACRYPT 2005*. Ed. by B. K. Roy. Vol. 3788. LNCS. Springer, Heidelberg, Dec. 2005, pp. 333–352. DOI: 10.1007/11593447_18.

[FLN07] P.-A. Fouque, G. Leurent, and P. Q. Nguyen. “Full Key-Recovery Attacks on HMAC/NMAC-MD4 and NMAC-MD5”. In: *CRYPTO 2007*. Ed. by A. Menezes. Vol. 4622. LNCS. Springer, Heidelberg, Aug. 2007, pp. 13–30. DOI: 10.1007/978-3-540-74143-5_2.

[Leu07] G. Leurent. “Message Freedom in MD4 and MD5 Collisions: Application to APOP”. In: *FSE 2007*. Ed. by A. Biryukov. Vol. 4593. LNCS. Springer, Heidelberg, Mar. 2007, pp. 309–328. DOI: 10.1007/978-3-540-74619-5_20.

[FL08] P.-A. Fouque and G. Leurent. “Cryptanalysis of a Hash Function Based on Quasi-cyclic Codes”. In: *CT-RSA 2008*. Ed. by T. Malkin. Vol. 4964. LNCS. Springer, Heidelberg, Apr. 2008, pp. 19–35. DOI: 10.1007/978-3-540-79263-5_2.

[Leu08a] G. Leurent. “MD4 is Not One-Way”. In: *FSE 2008*. Ed. by K. Nyberg. Vol. 5086. LNCS. Springer, Heidelberg, Feb. 2008, pp. 412–428. DOI: 10.1007/978-3-540-71039-4_26.

[FLR+09] P.-A. Fouque, G. Leurent, D. Réal, and F. Valette. “Practical Electromagnetic Template Attack on HMAC”. In: *CHES 2009*. Ed. by C. Clavier and K. Gaj. Vol. 5747. LNCS. Springer, Heidelberg, Sept. 2009, pp. 66–80. DOI: 10.1007/978-3-642-04138-9_6.

[LN09] G. Leurent and P. Q. Nguyen. “How Risky Is the Random-Oracle Model?” In: *CRYPTO 2009*. Ed. by S. Halevi. Vol. 5677. LNCS. Springer, Heidelberg, Aug. 2009, pp. 445–464. DOI: 10.1007/978-3-642-03356-8_26.

[BDL+10] C. Bouillaguet, O. Dunkelman, G. Leurent, and P.-A. Fouque. “Another Look at Complementation Properties”. In: *FSE 2010*. Ed. by S. Hong and T. Iwata. Vol. 6147. LNCS. Springer, Heidelberg, Feb. 2010, pp. 347–364. DOI: 10.1007/978-3-642-13858-4_20.

[GLM+10] P. Gauravaram, G. Leurent, F. Mendel, M. Naya-Plasencia, T. Peyrin, C. Rechberger, and M. Schläffer. "Cryptanalysis of the 10-Round Hash and Full Compression Function of SHAvite-3–512". In: *AFRICACRYPT 10*. Ed. by D. J. Bernstein and T. Lange. Vol. 6055. LNCS. Springer, Heidelberg, May 2010, pp. 419–436. DOI: 10.1007/978-3-642-12678-9_25.

[Leu10] G. Leurent. "Practical Key Recovery Attack against Secret-IV Edon-R". In: *CT-RSA 2010*. Ed. by J. Pieprzyk. Vol. 5985. LNCS. Springer, Heidelberg, Mar. 2010, pp. 334–349. DOI: 10.1007/978-3-642-11925-5_23.

[NRA+10] M. Naya-Plasencia, A. Röck, J.-P. Aumasson, Y. Laigle-Chapuy, G. Leurent, W. Meier, and T. Peyrin. "Cryptanalysis of ESSENCE". In: *FSE 2010*. Ed. by S. Hong and T. Iwata. Vol. 6147. LNCS. Springer, Heidelberg, Feb. 2010, pp. 134–152. DOI: 10.1007/978-3-642-13858-4_8.

[BDL+11] C. Bouillaguet, O. Dunkelman, G. Leurent, and P.-A. Fouque. "Attacks on Hash Functions Based on Generalized Feistel: Application to Reduced-Round Lesamnta and SHAvite-3512". In: *SAC 2010*. Ed. by A. Biryukov, G. Gong, and D. R. Stinson. Vol. 6544. LNCS. Springer, Heidelberg, Aug. 2011, pp. 18–35. DOI: 10.1007/978-3-642-19574-7_2.

[BFL11] C. Bouillaguet, P.-A. Fouque, and G. Leurent. "Security Analysis of SIMD". In: *SAC 2010*. Ed. by A. Biryukov, G. Gong, and D. R. Stinson. Vol. 6544. LNCS. Springer, Heidelberg, Aug. 2011, pp. 351–368. DOI: 10.1007/978-3-642-19574-7_24.

[LT11] G. Leurent and S. S. Thomsen. "Practical Near-Collisions on the Compression Function of BMW". In: *FSE 2011*. Ed. by A. Joux. Vol. 6733. LNCS. Springer, Heidelberg, Feb. 2011, pp. 238–251. DOI: 10.1007/978-3-642-21702-9_14.

[BDF+12] C. Bouillaguet, O. Dunkelman, P.-A. Fouque, and G. Leurent. "New Insights on Impossible Differential Cryptanalysis". In: *SAC 2011*. Ed. by A. Miri and S. Vaudenay. Vol. 7118. LNCS. Springer, Heidelberg, Aug. 2012, pp. 243–259. DOI: 10.1007/978-3-642-28496-0_15.

[KLR12] D. Khovratovich, G. Leurent, and C. Rechberger. "Narrow-Bicliques: Cryptanalysis of Full IDEA". In: *EUROCRYPT 2012*. Ed. by D. Pointcheval and T. Johansson. Vol. 7237. LNCS. Springer, Heidelberg, Apr. 2012, pp. 392–410. DOI: 10.1007/978-3-642-29011-4_24.

[Leu12] G. Leurent. "Analysis of Differential Attacks in ARX Constructions". In: *ASIACRYPT 2012*. Ed. by X. Wang and K. Sako. Vol. 7658. LNCS. Springer, Heidelberg, Dec. 2012, pp. 226–243. DOI: 10.1007/978-3-642-34961-4_15.

[LR12] G. Leurent and A. Roy. "Boomerang Attacks on Hash Function Using Auxiliary Differentials". In: *CT-RSA 2012*. Ed. by O. Dunkelman. Vol. 7178. LNCS. Springer, Heidelberg, Feb. 2012, pp. 215–230. DOI: 10.1007/978-3-642-27954-6_14.

[BLR13] A. Biryukov, G. Leurent, and A. Roy. "Cryptanalysis of the "Kindle" Cipher". In: *SAC 2012*. Ed. by L. R. Knudsen and H. Wu. Vol. 7707. LNCS. Springer, Heidelberg, Aug. 2013, pp. 86–103. DOI: 10.1007/978-3-642-35999-6_7.

[Leu13] G. Leurent. "Construction of Differential Characteristics in ARX Designs Application to Skein". In: *CRYPTO 2013, Part I*. Ed. by R. Canetti and J. A. Garay. Vol. 8042. LNCS. Springer, Heidelberg, Aug. 2013, pp. 241–258. DOI: 10.1007/978-3-642-40041-4_14.

[LPW13] G. Leurent, T. Peyrin, and L. Wang. “New Generic Attacks against Hash-Based MACs”. In: *ASIACRYPT 2013, Part II*. Ed. by K. Sako and P. Sarkar. Vol. 8270. LNCS. Springer, Heidelberg, Dec. 2013, pp. 1–20. DOI: 10.1007/978-3-642-42045-0_1.

[BGL+14] H. Brenner, L. Gaspar, G. Leurent, A. Rosen, and F.-X. Standaert. “FPGA Implementations of SPRING - And Their Countermeasures against Side-Channel Attacks”. In: *CHES 2014*. Ed. by L. Batina and M. Robshaw. Vol. 8731. LNCS. Springer, Heidelberg, Sept. 2014, pp. 414–432. DOI: 10.1007/978-3-662-44709-3_23.

[DL14] I. Dinur and G. Leurent. “Improved Generic Attacks against Hash-Based MACs and HAIFA”. In: *CRYPTO 2014, Part I*. Ed. by J. A. Garay and R. Gennaro. Vol. 8616. LNCS. Springer, Heidelberg, Aug. 2014, pp. 149–168. DOI: 10.1007/978-3-662-44371-2_9.

[GLS14] L. Gaspar, G. Leurent, and F.-X. Standaert. “Hardware Implementation and Side-Channel Analysis of Lapin”. In: *CT-RSA 2014*. Ed. by J. Benaloh. Vol. 8366. LNCS. Springer, Heidelberg, Feb. 2014, pp. 206–226. DOI: 10.1007/978-3-319-04852-9_11.

[GJL+14] J. Guo, J. Jean, G. Leurent, T. Peyrin, and L. Wang. “The Usage of Counter Revisited: Second-Preimage Attack on New Russian Standardized Hash Function”. In: *SAC 2014*. Ed. by A. Joux and A. M. Youssef. Vol. 8781. LNCS. Springer, Heidelberg, Aug. 2014, pp. 195–211. DOI: 10.1007/978-3-319-13051-4_12.

[Leu14a] G. Leurent. “Cryptanalysis of WIDEA”. In: *FSE 2013*. Ed. by S. Moriai. Vol. 8424. LNCS. Springer, Heidelberg, Mar. 2014, pp. 39–51. DOI: 10.1007/978-3-662-43933-3_3.

[Leu14b] G. Leurent. “Time-Memory Trade-Offs for Near-Collisions”. In: *FSE 2013*. Ed. by S. Moriai. Vol. 8424. LNCS. Springer, Heidelberg, Mar. 2014, pp. 205–218. DOI: 10.1007/978-3-662-43933-3_11.

[BBL+15] A. Banerjee, H. Brenner, G. Leurent, C. Peikert, and A. Rosen. “SPRING: Fast Pseudorandom Functions from Rounded Ring Products”. In: *FSE 2014*. Ed. by C. Cid and C. Rechberger. Vol. 8540. LNCS. Springer, Heidelberg, Mar. 2015, pp. 38–57. DOI: 10.1007/978-3-662-46706-0_3.

[FLS15] T. Fuhr, G. Leurent, and V. Suder. “Collision Attacks Against CAESAR Candidates - Forgery and Key-Recovery Against AEZ and Marble”. In: *ASIACRYPT 2015, Part II*. Ed. by T. Iwata and J. H. Cheon. Vol. 9453. LNCS. Springer, Heidelberg, Nov. 2015, pp. 510–532. DOI: 10.1007/978-3-662-48800-3_21.

[GLS+15] V. Grosso, G. Leurent, F.-X. Standaert, and K. Varici. “LS-Designs: Bitslice Encryption for Efficient Masked Software Implementations”. In: *FSE 2014*. Ed. by C. Cid and C. Rechberger. Vol. 8540. LNCS. Springer, Heidelberg, Mar. 2015, pp. 18–37. DOI: 10.1007/978-3-662-46706-0_2.

[LW15] G. Leurent and L. Wang. “The Sum Can Be Weaker Than Each Part”. In: *EUROCRYPT 2015, Part I*. Ed. by E. Oswald and M. Fischlin. Vol. 9056. LNCS. Springer, Heidelberg, Apr. 2015, pp. 345–367. DOI: 10.1007/978-3-662-46800-5_14.

[BL16a] K. Bhargavan and G. Leurent. “On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN”. In: *ACM CCS 2016*. Ed. by E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi. ACM Press, Oct. 2016, pp. 456–467. DOI: 10.1145/2976749.2978423.

[BL16b] K. Bhargavan and G. Leurent. "Transcript Collision Attacks: Breaking Authentication in TLS, IKE and SSH". In: *NDSS 2016*. The Internet Society, Feb. 2016. DOI: 10.14722/ndss.2016.23418.

[BLP16] A. Biryukov, G. Leurent, and L. Perrin. "Cryptanalysis of Feistel Networks with Secret Round Functions". In: *SAC 2015*. Ed. by O. Dunkelman and L. Keliher. Vol. 9566. LNCS. Springer, Heidelberg, Aug. 2016, pp. 102–121. DOI: 10.1007/978-3-319-31301-6_6.

[BCL+16] C. Boura, A. Chakraborti, G. Leurent, G. Paul, D. Saha, H. Soleimany, and V. Suder. "Key Recovery Attack Against 2.5-Round π-Cipher". In: *FSE 2016*. Ed. by T. Peyrin. Vol. 9783. LNCS. Springer, Heidelberg, Mar. 2016, pp. 535–553. DOI: 10.1007/978-3-662-52993-5_27.

[CDL16] A. Canteaut, S. Duval, and G. Leurent. "Construction of Lightweight S-Boxes Using Feistel and MISTY Structures". In: *SAC 2015*. Ed. by O. Dunkelman and L. Keliher. Vol. 9566. LNCS. Springer, Heidelberg, Aug. 2016, pp. 373–393. DOI: 10.1007/978-3-319-31301-6_22.

[KLL+16a] M. Kaplan, G. Leurent, A. Leverrier, and M. Naya-Plasencia. "Breaking Symmetric Cryptosystems Using Quantum Period Finding". In: *CRYPTO 2016, Part II*. Ed. by M. Robshaw and J. Katz. Vol. 9815. LNCS. Springer, Heidelberg, Aug. 2016, pp. 207–237. DOI: 10.1007/978-3-662-53008-5_8.

[KLL+16b] M. Kaplan, G. Leurent, A. Leverrier, and M. Naya-Plasencia. "Quantum Differential and Linear Cryptanalysis". In: *IACR Trans. Symm. Cryptol.* 2016.1 (2016), pp. 71–94. ISSN: 2519-173X. DOI: 10.13154/tosc.v2016.i1.71-94.

[Leu16a] G. Leurent. "Differential Forgery Attack Against LAC". In: *SAC 2015*. Ed. by O. Dunkelman and L. Keliher. Vol. 9566. LNCS. Springer, Heidelberg, Aug. 2016, pp. 217–224. DOI: 10.1007/978-3-319-31301-6_13.

[Leu16b] G. Leurent. "Improved Differential-Linear Cryptanalysis of 7-Round Chaskey with Partitioning". In: *EUROCRYPT 2016, Part I*. Ed. by M. Fischlin and J.-S. Coron. Vol. 9665. LNCS. Springer, Heidelberg, May 2016, pp. 344–371. DOI: 10.1007/978-3-662-49890-3_14.

[AEL+18] T. Ashur, M. Eichlseder, M. M. Lauridsen, G. Leurent, B. Minaud, Y. Rotella, Y. Sasaki, and B. Viguier. "Cryptanalysis of MORUS". In: *ASIACRYPT 2018, Part II*. Ed. by T. Peyrin and S. Galbraith. Vol. 11273. LNCS. Springer, Heidelberg, Dec. 2018, pp. 35–64. DOI: 10.1007/978-3-030-03329-3_2.

[DL18] S. Duval and G. Leurent. "MDS Matrices with Lightweight Circuits". In: *IACR Trans. Symm. Cryptol.* 2018.2 (2018), pp. 48–78. ISSN: 2519-173X. DOI: 10.13154/tosc.v2018.i2.48-78.

[LNS18] G. Leurent, M. Nandi, and F. Sibleyras. "Generic Attacks Against Beyond-Birthday-Bound MACs". In: *CRYPTO 2018, Part I*. Ed. by H. Shacham and A. Boldyreva. Vol. 10991. LNCS. Springer, Heidelberg, Aug. 2018, pp. 306–336. DOI: 10.1007/978-3-319-96884-1_11.

[LS18] G. Leurent and F. Sibleyras. "The Missing Difference Problem, and Its Applications to Counter Mode Encryption". In: *EUROCRYPT 2018, Part II*. Ed. by J. B. Nielsen and V. Rijmen. Vol. 10821. LNCS. Springer, Heidelberg, Apr. 2018, pp. 745–770. DOI: 10.1007/978-3-319-78375-8_24.

[LP19] G. Leurent and T. Peyrin. “From Collisions to Chosen-Prefix Collisions Application to Full SHA-1”. In: *EUROCRYPT 2019, Part III*. Ed. by Y. Ishai and V. Rijmen. Vol. 11478. LNCS. Springer, Heidelberg, May 2019, pp. 527–555. DOI: 10.1007/978-3-030-17659-4_18.

[LS19] G. Leurent and F. Sibleyras. “Low-Memory Attacks Against Two-Round Even-Mansour Using the 3-XOR Problem”. In: *CRYPTO 2019, Part II*. Ed. by A. Boldyreva and D. Micciancio. Vol. 11693. LNCS. Springer, Heidelberg, Aug. 2019, pp. 210–235. DOI: 10.1007/978-3-030-26951-7_8.

[BDL20] A. Bariant, N. David, and G. Leurent. “Cryptanalysis of Forkciphers”. In: *IACR Trans. Symm. Cryptol.* 2020.1 (2020), pp. 233–265. ISSN: 2519-173X. DOI: 10.13154/tosc.v2020.i1.233-265.

[BBB+20] D. Bellizia, F. Berti, O. Bronchain, G. Cassiers, S. Duval, C. Guo, G. Leander, G. Leurent, I. Levi, C. Momin, O. Pereira, T. Peters, F.-X. Standaert, B. Udvarhelyi, and F. Wiemer. “Spook: Sponge-Based Leakage-Resistant Authenticated Encryption with a Masked Tweakable Block Cipher”. In: *IACR Trans. Symm. Cryptol.* 2020.S1 (2020), pp. 295–349. ISSN: 2519-173X. DOI: 10.13154/tosc.v2020.iS1.295-349.

[BCD+20] T. Beyne, A. Canteaut, I. Dinur, M. Eichlseder, G. Leander, G. Leurent, M. Naya-Plasencia, L. Perrin, Y. Sasaki, Y. Todo, and F. Wiemer. “Out of Oddity - New Cryptanalytic Techniques Against Symmetric Primitives Optimized for Integrity Proof Systems”. In: *CRYPTO 2020, Part III*. Ed. by D. Micciancio and T. Ristenpart. Vol. 12172. LNCS. Springer, Heidelberg, Aug. 2020, pp. 299–328. DOI: 10.1007/978-3-030-56877-1_11.

[CDL+20] A. Canteaut, S. Duval, G. Leurent, M. Naya-Plasencia, L. Perrin, T. Pornin, and A. Schrottenloher. “Saturnin: a suite of lightweight symmetric algorithms for post-quantum security”. In: *IACR Trans. Symm. Cryptol.* 2020.S1 (2020), pp. 160–207. ISSN: 2519-173X. DOI: 10.13154/tosc.v2020.iS1.160-207.

[DL20] I. Dinur and G. Leurent. “Preface to Volume 2020, Special Issue on Designs for the NIST Lightweight Standardisation Process”. In: *IACR Trans. Symm. Cryptol.* 2020.S1 (2020), pp. 1–4. ISSN: 2519-173X. DOI: 10.13154/tosc.v2020.iS1.1-4.

[FLN+20] A. Flórez-Gutiérrez, G. Leurent, M. Naya-Plasencia, L. Perrin, A. Schrottenloher, and F. Sibleyras. “New Results on Gimli: Full-Permutation Distinguishers and Improved Collisions”. In: *ASIACRYPT 2020, Part I*. Ed. by S. Moriai and H. Wang. Vol. 12491. LNCS. Springer, Heidelberg, Dec. 2020, pp. 33–63. DOI: 10.1007/978-3-030-64837-4_2.

[LP20] G. Leurent and T. Peyrin. “SHA-1 is a Shambles: First Chosen-Prefix Collision on SHA-1 and Application to the PGP Web of Trust”. In: *USENIX Security 2020*. Ed. by S. Capkun and F. Roesner. USENIX Association, Aug. 2020, pp. 1839–1856. URL: https://www.usenix.org/system/files/sec20-leurent.pdf.

[LS20] G. Leurent and Y. Sasaki. “Preface to Volume 2020, Issue 1”. In: *IACR Trans. Symm. Cryptol.* 2020.1 (2020), pp. 1–5. ISSN: 2519-173X. DOI: 10.13154/tosc.v2020.i1.1-5.

[LLW+20] Y. Li, G. Leurent, M. Wang, W. Wang, G. Zhang, and Y. Liu. “Universal Forgery Attack Against GCM-RUP”. In: *CT-RSA 2020*. Ed. by S. Jarecki. Vol. 12006. LNCS. Springer, Heidelberg, Feb. 2020, pp. 15–34. DOI: 10.1007/978-3-030-40186-3_2.

[BDL+21] C. Beierle, P. Derbez, G. Leander, G. Leurent, H. Raddum, Y. Rotella, D. Rupprecht, and L. Stennes. "Cryptanalysis of the GPRS Encryption Algorithms GEA-1 and GEA-2". In: *EUROCRYPT 2021, Part II*. Ed. by A. Canteaut and F.-X. Standaert. Vol. 12697. LNCS. Springer, Heidelberg, Oct. 2021, pp. 155–183. DOI: 10.1007/978-3-030-77886-6_6.

[BBC+21] R. Bhaumik, X. Bonnetain, A. Chailloux, G. Leurent, M. Naya-Plasencia, A. Schrottenloher, and Y. Seurin. "QCB: Efficient Quantum-Secure Authenticated Encryption". In: *ASIACRYPT 2021, Part I*. Ed. by M. Tibouchi and H. Wang. Vol. 13090. LNCS. Springer, Heidelberg, Dec. 2021, pp. 668–698. DOI: 10.1007/978-3-030-92062-3_23.

[BLN+21] X. Bonnetain, G. Leurent, M. Naya-Plasencia, and A. Schrottenloher. "Quantum Linearization Attacks". In: *ASIACRYPT 2021, Part I*. Ed. by M. Tibouchi and H. Wang. Vol. 13090. LNCS. Springer, Heidelberg, Dec. 2021, pp. 422–452. DOI: 10.1007/978-3-030-92062-3_15.

[CKL+21] A. Chattopadhyay, M. Khairallah, G. Leurent, Z. Najm, T. Peyrin, and V. Velichkov. "On the Cost of ASIC Hardware Crackers: A SHA-1 Case Study". In: *CT-RSA 2021*. Ed. by K. G. Paterson. Vol. 12704. LNCS. Springer, Heidelberg, May 2021, pp. 657–681. DOI: 10.1007/978-3-030-75539-3_27.

[DL21] I. Dinur and G. Leurent. "Preface to Volume 2021, Issue 1". In: *IACR Trans. Symm. Cryptol.* 2021.1 (2021), pp. 1–4. ISSN: 2519-173X. DOI: 10.46586/tosc.v2021.i1.1-4.

[LP21] G. Leurent and C. Pernot. "New Representations of the AES Key Schedule". In: *EUROCRYPT 2021, Part I*. Ed. by A. Canteaut and F.-X. Standaert. Vol. 12696. LNCS. Springer, Heidelberg, Oct. 2021, pp. 54–84. DOI: 10.1007/978-3-030-77870-5_3.

[LPS21] G. Leurent, C. Pernot, and A. Schrottenloher. "Clustering Effect in Simon and Simeck". In: *ASIACRYPT 2021, Part I*. Ed. by M. Tibouchi and H. Wang. Vol. 13090. LNCS. Springer, Heidelberg, Dec. 2021, pp. 272–302. DOI: 10.1007/978-3-030-92062-3_10.

[BBL+22] A. Bariant, C. Bouvier, G. Leurent, and L. Perrin. "Algebraic Attacks against Some Arithmetization-Oriented Primitives". In: *IACR Trans. Symm. Cryptol.* 2022.3 (2022), pp. 73–101. DOI: 10.46586/tosc.v2022.i3.73-101.

[BL23] A. Bariant and G. Leurent. "Truncated Boomerang Attacks and Application to AES-Based Ciphers". In: *EUROCRYPT 2023, Part IV*. Ed. by C. Hazay and M. Stam. Vol. 14007. LNCS. Springer, Heidelberg, Apr. 2023, pp. 3–35. DOI: 10.1007/978-3-031-30634-1_1.

Research Reports

[LBF08] G. Leurent, C. Bouillaguet, and P.-A. Fouque. *SIMD Is a Message Digest*. Submitted to the SHA-3 competition. 2008. URL: https://who.rocq.inria.fr/Gaetan.Leurent/simd.html.

[GLS+14] V. Grosso, G. Leurent, F.-X. Standaert, K. Varici, F. Durvaux, L. Gaspar, and S. Kerckhof. *SCREAM & iSCREAM: Side-Channel Resistant Authenticated Encryption with Masking*. Submitted to the CAESAR competition. Mar. 2014. URL: http://perso.uclouvain.be/fstandae/SCREAM/.

[BDD+17] X. Bonnetain, P. Derbez, S. Duval, G. L. Jérémy Jean, B. Minaud, and V. Suder. *AEZ forgeries*. Message on the `crypto-competition` mailing list. Presented at the FSE 2017 rump session: https://iacr.org/workshops/fse2017/slides/Rump-02.pdf. Mar. 2017. URL: https://groups.google.com/g/crypto-competitions/c/YWkEZkLKr9E/m/75LG5KmeBQAJ.

[LMP+23] G. Leurent, B. Mennink, K. Pietrzak, and V. Rijmen. *Analysis of MinRoot: Public report*. Tech. rep. Ethereum Foundation, Sept. 2023. URL: https://crypto.ethereum.org/events/minrootanalysis2023.pdf.

Outreach

[LN18] G. Leurent and M. Naya-Plasencia. "La fagilité inattendue du chiffrement symétrique". In: *La Recherche*. 514. Sophia Publications, Nov. 2018, pp. 54–56.

[BCC+21] X. Bonnetain, A. Canteaut, V. Cortier, P. Gaudry, L. Hirschi, S. Kremer, S. Lacour, M. Lequesne, G. Leurent, L. Perrin, A. Schrottenloher, E. Thomé, S. Vaudenay, and C. Vuillot. "Le traçage anonyme, dangereux oxymore – Analyse de risques à destination des non-spécialistes". In: *Le droit face au coronavirus*. Ed. by Club des juristes. LexisNexis, Dec. 2021. ISBN: 978-2-7110-3639-4.

1. Introduction

Cryptography enables secure communication in the presence of an adversary, with tools to guarantee the confidentiality and authenticity of messages, and to authenticate users. It is a vital part of the today's digital world, where we exchange private information over the internet, we buy goods and services on the web, industrial control systems are networked, banks settle financial transactions online, and state agencies communicate remotely. Indeed, we have strong evidence that adversaries are trying to intercept communications and to penetrate computer systems, from cybercriminals to state agencies performing massive surveillance and to private companies collecting personal data for targeted advertising.

Cryptographic schemes are designed with a bottom-up approach, starting from primitives that operate on small messages and achieve a well-defined security notion (such as being hard to invert). Then, primitives are used inside a mode of operation (or mode) and a protocol to deal with session establishment and arbitrary messages.

In order to allow security evaluation of the schemes, the algorithms must be published, and their security must rely on a small piece of information that is different for each user, and kept secret: the *key*. These principles have been established as early as 1883 by Auguste Kerckhoffs:

> “
> 1. *Le système doit être matériellement, sinon mathématiquement, indéchiffrable;*
> 2. *Il faut qu'il n'exige pas le secret, et qu'il puisse sans inconvénient tomber entre les mains de l'ennemi;*
> 3. *La clef doit pouvoir en être communiquée et retenue sans le secours de notes écrites, et être changée ou modifiée au gré des correspondants;*
> 4. *Il faut qu'il soit applicable à la correspondance télégraphique;*
> 5. *Il faut qu'il soit portatif, et que son maniement ou son fonctionnement n'exige pas le concours de plusieurs personnes;*
> 6. *Enfin, il est nécessaire, vu les circonstances qui en commandent l'application, que le système soit d'un usage facile, ne demandant ni tension d'esprit, ni la connaissance d'une longue série de règles à observer.* ”
>
> — Auguste Kerckhoffs, [Ker83]

1.1 Symmetric and asymmetric cryptography

The most natural cryptographic construction use the same key to encrypt a message and to decrypt it; it has been used since antiquity to protect military communications. Today, we denote the corresponding field as *symmetric cryptography* or *secret-key cryptography*.

Symmetric cryptography. Symmetric primitives are mostly dedicated designs that try to mix the key and the message in a complex way following the notions of confusion and diffusion formalized by Shannon in 1949 [Sha49]. We have four main types of primitives in

use today: block ciphers (such as AES [AES]), stream ciphers (such as Salsa20 [Ber08]), hash functions (such as the SHA standards [SHA1; SHA2; SHA3]), and cryptographic permutations (such as Keccak-f [SHA3]).

Symmetric primitives usually work on small blocks of fixed size (n bits, typically $n \in \{64, 128, 256\}$ for block ciphers, and $256 \leq n \leq 2048$ for hash functions and cryptographic permutations). Then, they are used inside a mode of operation to deal with arbitrary-length messages. A mode divides the message M into n-bit blocks m_i, and processes the blocks one by one through the primitive with various chaining rules. Standard modes of operation starting from a block cipher can provide encryption (CBC, CTR), authentication (CBC-MAC, PMAC), or authenticated encryption (GCM, OCB). Rogaway gives a good overview of the main block-cipher modes and their security [Rog11]. Similarly, the sponge construction and its variants turn a cryptographic permutation into a hash function, a stream cipher, or an authenticated encryption. Hash functions are typically considered a primitive, but they are actually built using a mode of operation (often Merkle-Damgård or HAIFA) from a smaller primitive: a compression function.

In the last years, an important trend is the design of *lightweight* primitives, for use in embedded devices with very limited power (medical implants, contactless smart cards, sensor nodes, ...), where standard cryptography is too expensive. In particular, the US National Institute of Standards and Technology (NIST) has recently organized a competition to select new standards in this field.[1]

Some industry applications use proprietary (non-public) algorithms, in particular for lightweight use cases where standard algorithms do not fit. In most cases, these algorithms turn out to be weak and can be broken in practice, such as CSS [Ste99] (used for the DRM system of DVDs), PC1 (used for the DRM system of Amazon ebooks) [BLR13], Keeloq [IKD+08; EKM+08] (used in car keys from multiple manufacturers), DST 40 [WMA+19] (also used in car keys) or MiFare [GKM+08] (used in transport card such as the Oyster card in London).

Asymmetric cryptography. On the other hand *asymmetric cryptography*, or *public-key cryptography* uses different keys to encrypt a message and to decrypt it. Therefore, the encryption key does not have to be kept secret, and can be widely distributed. This solves the problem of key distribution, which is a big drawback of symmetric cryptography.

The most widely used constructions today are based on hard problems from number theory: the Diffie-Hellman key exchange [DH76] relies on the hardness of computing discrete logarithms, while the RSA signature and encryption scheme [RSA78] relies on the hardness of factoring.

However, those constructions would be broken by a quantum computer, if such a device can be built. Indeed, Shor's algorithm [Sho94] computes discrete logarithms and factors integers in quantum polynomial time. Following recent advances in quantum information theory, there is a strong interest in constructions that rely on hard problems that are believed to resist quantum computers, such as code-based cryptosystems [McE78] or lattice-based cryptosystems [Ajt96; Reg05]. The NIST has recently organized a standardization process for this type of post-quantum algorithms.[2]

Hybrid systems. Many security systems rely on public-key cryptography for key establishment, but only secret-key cryptography can offer sufficient performance to process bulk messages. Therefore, in practice, we use hybrid schemes that combine public-key cryptography and symmetric cryptography.

[1] https://csrc.nist.gov/projects/lightweight-cryptography
[2] https://csrc.nist.gov/projects/post-quantum-cryptography

1.2 Evaluating cryptographic schemes

Since the seminal work of Goldwasser and Micali [GM84], provable security has become the accepted practice in cryptography. Proposals for new schemes and protocols are expected to come with a security reduction, showing that the system is secure under a reasonable security assumption, or security model, for the underlying primitive(s).

A security proof starts by defining a security notion, and bounds the probability (or advantage) of an adversary to break that security notion, depending on the resources she can use, and the security of the primitive. Provable security focuses on asymptotic security: a mode is secure if the bound decreases exponentially with n.

Concrete security. In order to better understand the security of cryptographic schemes, the "concrete security" [BDJ+97] approach writes explicit bounds on the security depending on the parameters of the system. This allows to select appropriate parameters. For instance, the security proofs of several `CBC-MAC` variants [BKR94; PR00] can be written as:

$$\mathrm{Adv}^{\mathrm{PRF}}_{\texttt{CBC-MAC-}E} \leq \mathrm{Adv}^{\mathrm{PRP}}_{E} + \frac{\sigma^2}{2^n},$$

where E is an n-bit block cipher, and σ the amount of data authenticated. This means that `CBC-MAC` is a secure family of pseudo-random functions (PRF) as long as $\sigma \ll 2^{n/2}$, and the block cipher is a secure pseudo-random permutation family (PRP).

In practice, the bounds of many modes of operation include a term in the order of $\sigma^2/2^n$, where σ is the total length of the queries made by the adversary. This means that the mode is only secure when $\sigma \ll 2^{n/2}$, which is called birthday-bound security.

Cryptanalysis. Cryptanalysis is another way to evaluate the security of cryptographic schemes, using a whole range of algorithmic knowledge to search for efficient attacks. There are classical techniques that apply to large classes of schemes (such as differential [BS91] and linear [Mat94] cryptanalysis), but cryptanalysis is still a difficult task because these techniques must be tailored to each scheme, and new techniques must often be devised.

Cryptanalysis is a crucial part of cryptographic research because we cannot prove unconditionally that a cipher is secure. In particular, we cannot prove the security of cryptographic primitives, because we cannot reduce their security to the security of a lower-level components. Therefore, a large, public cryptanalysis effort is of utmost importance. If such an effort unveils weaknesses in a cryptographic construction, it will be deprecated, hopefully before the weaknesses lead to practical breaks. Conversely, a continued cryptanalysis effort lasting over several years without finding security weaknesses is a good way to build trust in the security of a construction. For instance the AES standard is a widely available and well trusted block cipher, because numerous experts have tried to break it in the last 25 years.

1.3 Cryptography usage in practice

Cryptography is now used routinely in our daily lives. Most of the devices with wireless communication use some kind of encryption, because radio transmissions are easy to intercept: Wi-Fi uses WEP (based on RC4) or WPA (based on AES), cell phone communications are encrypted with Kasumi, SNOW 3G, AES, or ZUC, Bluetooth uses E0, cordless phones use the DECT Standard Cipher, etc. Cryptography is also widely used to authenticate legitimate users of a system: credit cards use public-key cryptography to prove to a reader that they are genuine; wireless keys fobs or cards are used to open cars, to access public transports, or to enter buildings; and passports contain an RFID chip to authenticate them.

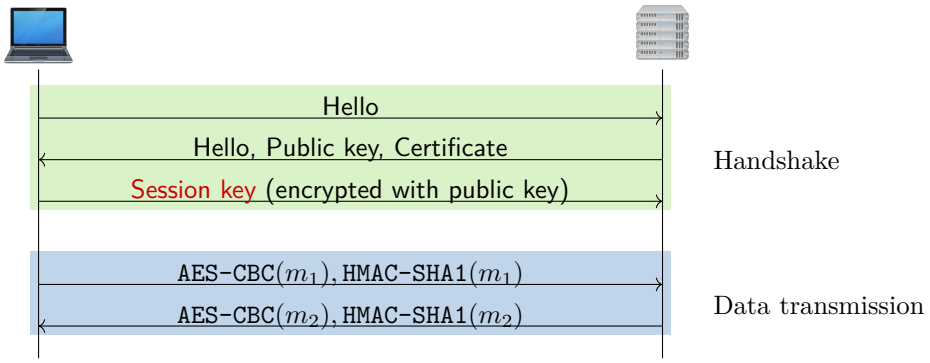


Figure 1.1: TLS: an example of secure channel establishment.

One of the most successful deployments of cryptography in practice is the SSL/TLS protocol, with more than 80% of pages loaded on the web now encrypted with TLS.[3] It is also a good example to understand what a real-world attacker can do.

The Transport Layer Security protocol (TLS). TLS is a transport protocol that can be used to encapsulate various internet protocols. It is widely used today on the internet, with most HTTP and IMAP connection secured by TLS (HTTP over TLS is denoted as HTTPS). TLS derives from SSL, originally designed by Netscape, with SSL 2.0 released in 1995 and SSL 3.0 in 1996. Development of the protocol was then taken over by the IETF, and later versions are denoted as TLS, with four version up to date: TLS 1.0 in 1999, TLS 1.1 in 2006, TLS 1.2 in 2008, and TLS 1.3 in 2018.

TLS is a hybrid protocol, using public-key cryptography for key establishment, and symmetric cryptography to encrypt and authenticate the actual data with the session key. When a client connects to a server, they start a handshake phase, where public-key cryptography is used to authenticate the server and to establish the session key, and parameters of the connection are negotiated (in particular the choice of ciphers to use). Figure 1.1 shows a typical example of TLS handshake and data transmission.

When used properly, TLS guarantees the identity of the server (and optionally the identity of the client), and protects the data from interception and modification.

The man-in-the-browser setting. When TLS is used to secure web connections, an attacker can use the man-in-the-browser setting to mount powerful attacks in practice (Figure 1.2). This setting was first considered in the BEAST attack [DR11], and assumes that:

1. The attacker has access to the network and can read and modify network packets.

   This is easy to realize when the victim is connected to an open Wi-Fi network, as used in many public spaces.

2. The victim is logged-in to a valuable website (such as a webmail, or an online bank).

   Since HTTP is a stateless protocol, the victim's browser uses an authentication token (usually a secret cookie) to identify queries to the website, and every HTTP query includes this token in a header.

3. The victim visits an attacker-controlled website.

[3] https://letsencrypt.org/stats/

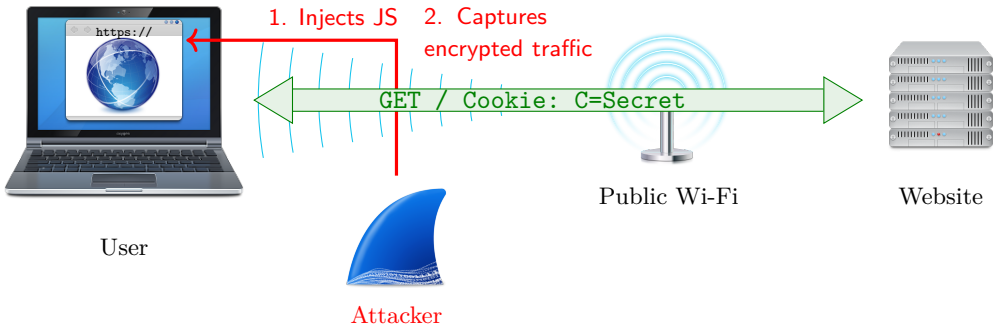


Figure 1.2: The man-in-the-browser setting.

```
var url = "https://10.0.0.1/index.html";
var xhr = new XMLHttpRequest;

while(true) {
    xhr.open("HEAD", url, false);
    xhr.withCredentials = true;
    xhr.send();
    xhr.abort();
}
```

Figure 1.3: Simple JavaScript code to encrypt chosen messages to a target website.

In practice, the attacker can use phishing, or modify a web page on the fly if the victim visits a non-encrypted website.

When those conditions are met, the attacker can use the malicious website to send arbitrary JavaScript code to the victim. This code will be executed by the victim's browser, and can make arbitrary queries to the target website (see Figure 1.3 for a concrete example). Since HTTP is stateless, every query includes the authentication token. The attacker cannot read the plaintext messages with the token or receive the answers from the target website, but she can ask for the encryption of chosen messages that include the token, and capture the corresponding ciphertexts on the network. This is a powerful attack scenario, showing that chosen-plaintext attacks are actually practical.

The attacker will try to exploit weaknesses of TLS to recover the authentication token. If she succeeds, she can use the token to authenticate as the victim on the target website, and extract all the information she wants.

The chosen-prefix secret-suffix model. In order to make attacks easier to explain, Hoang *et al.* have proposed a simple model to capture the type of attacks possible in the man-in-the-browser setting [HRR+15]. The attacker must recover a secret value S, given access to an oracle that computes the encryption of $M \parallel S$ for a chosen prefix M:

$$\mathcal{O} : M \mapsto \mathcal{E}(M \parallel S).$$

This is an important model because it corresponds to practical adversaries in the man-in-the-browser setting, even though chosen-plaintext attacks can seem artificial. It is also sufficiently generic to capture most attacks in practice. In particular, the attacker can vary the length of the prefix, and move the secret suffix across block boundaries.

Table 1.1: Symmetric cryptography usage on the web, according to different sources. Telemetry data [Moz23] is from Firefox Nightly; it represents the percentage of actual connections using each cipher. Internet scan data [Cen23] is from Censys [DAM+15]; it represents the percentage of public internet services that would use each cipher. The discrepancy in Chacha-Poly usage is due to different client-side preferences.

Source	Telemetry [Moz23] *per connection*	Internet scan [Cen23] *per service (host, port)*
Encryption		
AES-GCM	98.7%	61.8%
Chacha-Poly	0.6%	32.9%
AES-CBC	0.7%	5.2%
3DES-CBC	0.0001%	0.1%
MAC		
HMAC-SHA256	76.7%	67.9%
HMAC-SHA384	22.5%	26.9%
HMAC-SHA1	0.7%	5.2%

Moreover, if the attacker wins this game, there is a real impact, with plaintext recovery and access to a secure resource when the secret is an access token. We will use this setting when describing attacks against modes of operation, rather than the distinguishers used in security proofs.

Primitive usage. The vast majority of deployed cryptography uses public standards that have been well analyzed. Unfortunately, many old standards are known to be insecure, because of mistakes in their design or inadequate security level, but still have non-negligible usage. For instance, GSM encryption (using `A5/1` [BBK08]) and WEP (using `RC4` [FMS01; TWP08]) can be broken easily.

We give some statistics about current (late 2023) cryptography usage in the web (HTTPS traffic) in Table 1.1: we see that almost all traffic is secured with robust algorithms: AES-GCM or Chacha-Poly for encryption, HMAC-SHA256 or HMAC-SHA384 for authenticity. However, there is still some fraction of the traffic that uses HMAC-SHA1 and non-zero usage of 3DES (triple DES).

1.4 Cryptanalysis beyond primitives

Cryptanalysis techniques are mostly used to target primitives and assess their security. However, we can also apply those techniques to modes of operations and protocols.

1.4.1 Generic attacks

Generic attacks are attacks that don't use any specific property of the block cipher. A generic attack against a mode of operation will work for any instantiation with a concrete block cipher. While dedicated attacks using properties of the block cipher are more powerful in general, generic attacks uncover properties of the mode itself.

Typically, cryptanalysis is used to study the security of primitives, and upper layers (modes and protocols) are studied with security proofs. However, there are many cases where security proofs do not give a full picture of the security of a scheme, and studying generic attacks gives a better understanding of the security. This occurs in the following typical situations:

Some proofs are flawed: several high-profile modes of operations have been broken, even though they had been proven secure, such as `EAX'` [MLM+14], or `OCB2` [IIM+19]. Indeed, security proofs are very technical; it is easy to make a mistake, and hard to verify the correctness of a proof.

Some proofs are not tight: when there is a gap between the proof and the best attack, more analysis is needed. For instance, `CENC` [Iwa06] and `SUM-ECBC` [Yas10] were initially proven secure up to $2^{2n/3}$ queries, but no attack faster than 2^n was known; after further analysis, it turned out that the security of `CENC` is essentially 2^n [IMV16], while the security of `SUM-ECBC` is about $2^{3n/4}$ [LNS18; KLL20].

The security depends on the model used: security proofs make assumption about the power of the adversary (such as knowing or choosing the plaintext), and this has a large impact on the achieved security. For instance encryption without authentication is secure against passive adversaries, but insecure against active adversaries. More recently, cryptography have been studied in a quantum setting, and many standardized MACs proven secure in the classical setting were shown insecure [KLL+16a].

Modes with similar proofs can have different security: even if the proof is tight, the security degradation outside the domain of the proof can range from low-impact distinguishers to devastating key-recovery. For instance, `PMAC` is secure up to $2^{n/2}$ queries [Rog04], and there is a matching forgery attack [LKS+06]. However, small variations of `PMAC` become susceptible to a key-recovery attack with $2^{n/2}$ queries, even though the security proof is the same [FLS15].

Understanding attacks helps us design better modes: Understanding generic attacks also helps to design more secure modes. For instance, the generic attack by Preneel and van Oorschot [Pv95] against iterated MACs explains the constructions used for MACs with security beyond the birthday bound: `RMAC` [JJV02] and the Wegman-Carter MAC [WC81] are randomized, while `SUM-ECBC` [Yas10] and `PMAC+` [Yas11] use a $2n$-bit internal state.

Real-world usage sometimes violates the assumptions of the proofs: SHA-1 has been broken decades ago, but it has stayed in use for a long time in protocols whose proofs assumed collision-resistance, because deprecation takes time, and attacks against primitives do not directly translate to attacks on real-world protocols (even when they invalidate the proofs). Block ciphers with 64-bit blocks (such as 3DES) have also been used with an amount of data that violates the assumptions of the proofs [BL16a].

1.4.2 Real-world impact of cryptanalysis

Weak cryptography can be used long after weaknesses have been found by the academic community. Unfortunately, public demonstrations of high-profile attacks seem to be the only way to convince the industry to move away from widely used standards with known weaknesses, as expressed by AlFardan and Paterson:

> *“ Our long-term aim is to ensure that weak encryption options are eliminated from TLS, to the eventual benefit of all users of TLS. Experience shows that the only way to make this happen is to make the attacks as powerful as possible and build proof-of-concept implementations of them. ”*
>
> — Nadhem AlFardan and Kenny Paterson, [AP13a]

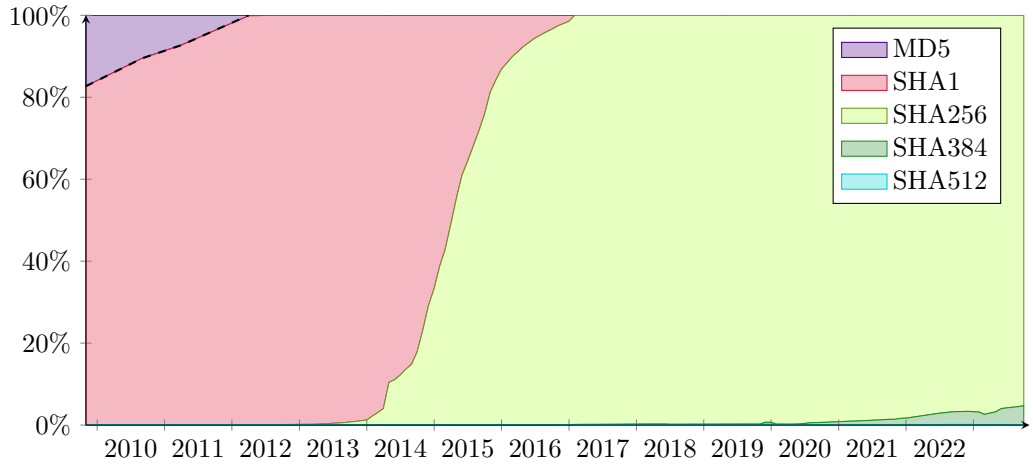


Figure 1.4: Hash function usage in certificates from top websites (150 000 certificates). Source: [HBK+11] (2009-2011), https://www.ssllabs.com/ssl-pulse/ (after 2012).

For instance, den Boer and Bosselaers found weaknesses in MD5 as early as 1993, with a collision attack on the compression function [dB94]. Even after the full collision attack of Wang *et al.* in 2004 [WY05], many users argued that collisions only had a limited impact on applications. Indeed, the attacker must choose *both messages* to exploit a collision, and she has little control over the values of the actual colliding messages, where the differences are inserted. In particular, MD5 continued to be used to sign website certificates, as seen in Figure 1.4. It took a public demonstration of the creation of a rogue certificate in 2009 to eliminate MD5 from certificate authorities [SSA+09]. The same story happened with SHA-1: a collision attack has been known since 2005 [WYY05], but due to its high computational complexity, it was only implemented in practice in 2017, using a large GPU cluster [SBK+17]. SHA-1 has been widely used until 2015, and there was still non-negligible usage of SHA-1 in 2020 (outside web certificates), when we demonstrated a practical impersonation attack against the PGP/GnuPG Web-of-Trust using weaknesses of SHA-1.

As another example, several attacks against TLS have been announced in the last years, and forced the deployment of countermeasures, even though the underlying weaknesses were known for a long time. The BEAST attack [DR11] is due to the use of predictable IVs, which was known to be an issue since at least 1995.[4] The CRIME attack [RD12] uses compression, which was seen as a problem in 2002 [Kel02]. The Lucky Thirteen [AP13b] and POODLE [MDK14] attacks use a padding oracle, as defined by Vaudenay [Vau02]. In 2013 and 2015, new attacks against RC4 use in TLS [ABP+13; GPv15; VP15] used biases in the keystream that have been known since 2000 [FM01; MS02]. Many of these attacks are based on the man-in-the-browser attacker model.

Algorithm negotiation and downgrade attacks. Another issue in practice is the possibility of downgrade attacks when a client and a server negotiate the protocol to use and its parameters (in particular algorithms and key sizes). Algorithm negotiation is designed so that the client and server agree on the *best* algorithm that they both support. However, in some cases an attacker can force them to use the *worst* algorithm that they support. TLS supports a huge range of ciphersuites, including modern authenticated encryption algorithms such as AES-GCM, and obsolete algorithms like DES and RC4, but the obsolete algorithms are often enabled to support legacy clients.

[4] http://web.cs.ucdavis.edu/~rogaway/papers/draft-rogaway-ipsec-comments-00.txt

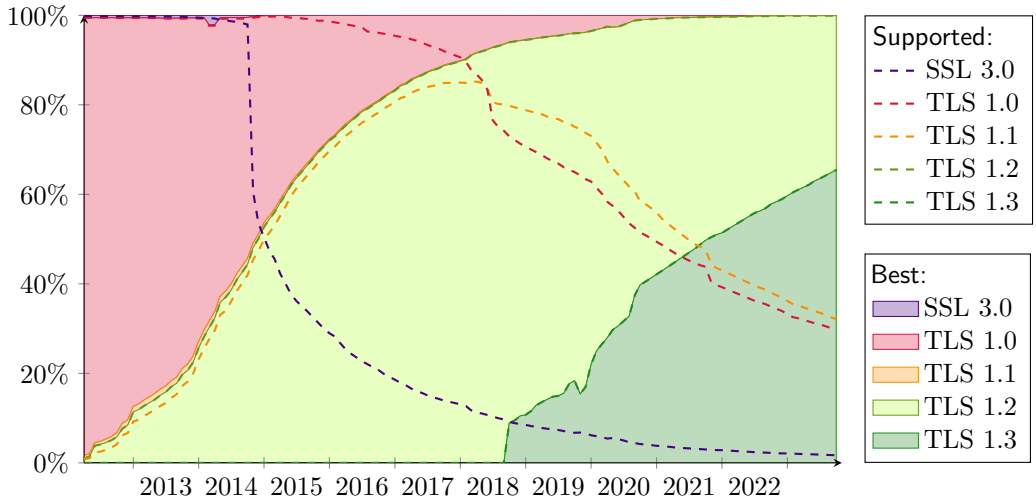


Figure 1.5: SSL/TLS versions supported by top websites (150 000 websites).
Source: https://www.ssllabs.com/ssl-pulse/.

This has been exploited in many different attacks over the years, leading deprecation of older standards and ciphers one by one. The SSLStrip [Mar09] removes redirections from HTTP to HTTPS, so that the client keeps using unsecured HTTP even if the server wants to use HTTPS. The POODLE [MDK14] attacks exploits insecure version fallback from TLS to SSL 3.0. The FREAK [BBD+15] and Logjam [ABD+15] attacks exploit export-grade ciphers that are weak on purpose, but that many clients and servers still supported. Downgrade attacks are also a concern in telephony network, where phones will switch to the insecure GSM network if more secure networks (3G, 4G, 5G) are jammed.

In particular, some attacks exploiting weaknesses of cryptographic algorithms are possible as long as the algorithms are *supported* by implementations, even if they are not used in normal conditions.

In recent years, there has been significant improvements, with TLS 1.3 disabling many legacy algorithms (in particular MD5 and SHA-1), and including a robust downgrade-detection mechanism [BBF+16]. Web browsers are also disabling TLS 1.0 and TLS 1.1, which don't offer modern authenticated encryption algorithms and depend on SHA-1.

1.5 Prelude: the birthday bound

The security of the vast majority of cryptosystems is affected by the presence of collisions. This is a well-known issue for hash functions: due to the birthday paradox we expect to find collisions in an n-bit hash function with roughly $2^{n/2}$ queries. Intuitively, with $2^{n/2}$ queries, there are about 2^{n-1} pairs of queries, and each pair is a collision with probability 2^{-n} if it behaves randomly. The pairs are not independent, but assuming independence gives a simple explanation that is quite close to the truth.

More formally, we consider t independent random variables in $\{0,1\}^n$. The probability that they are all distinct is

$$\begin{aligned}\Pr[\text{no collision}] &= 1 \times \frac{2^n-1}{2^n} \times \frac{2^n-2}{2^n} \times \frac{2^n-3}{2^n} \times \ldots = \prod_{i=0}^{t-1} 1 - \frac{i}{2^n} \\ &\approx \exp\left(\frac{-t^2}{2^{n+1}}\right)\end{aligned}$$

A variant with two sets of random variables is also useful for cryptographic applications. Given two random subsets $\mathcal{S}, \mathcal{T} \subset \{0,1\}^n$, they have a non-empty intersection with high probability as soon as $|\mathcal{S}| \cdot |\mathcal{T}| \gtrapprox 2^n$.

We note that the theoretical analysis assumes independent random variables. In practice, we often apply it to cryptographic functions that are not actually random, but deviation from the random behavior would typically be a sign of a deeper weakness in the cryptosystem. Moreover, experiments can validate this heuristic analysis.

1.5.1 Finding collisions efficiently

The birthday paradox shows that collisions exist after roughly $2^{n/2}$ evaluations of a random function h. This defines the security level of an ideal hash function in the information-theoretic model, where we only count oracle calls, and we don't consider the additional computations required to locate the collision.

In practice, we find collisions efficiently by building a list of $2^{n/2}$ inputs and outputs of the hash function, and sorting it according to the hash output. Then we locate the collisions by traversing the list. This requires $\mathcal{O}(n \times 2^{n/2})$ operations and $\mathcal{O}(n)$ memory. There are more advanced techniques with almost the same time complexity using very little memory. The rho method (ρ) by Pollard [Pol75] considers iterates of the function h: $x_i = h(x_{i-1})$. After some iterations, the values enter a cycle, and the cycle entry is a collision. Using Floyd's tortoise and hare algorithm, the cycle entry is detected with constant memory: a collision is found with $\mathcal{O}(2^{n/2})$ operations and $\mathcal{O}(1)$ memory.

The variant of van Oorschot and Wiener [vW99] has essentially the same complexity, but uses distinguished point to allow efficient parallelization. Moreover, it generalizes well to the search for multiple collisions: finding m collisions requires a memory $\mathcal{O}(m)$ and time $\mathcal{O}(2^{n/2} \times \sqrt{m})$. The time complexity matches the information-theoretic lower bound up to a constant factor.

When looking for a collision between two sets $\mathcal{S}$ and $\mathcal{T}$, we sort both of them and traverse them simultaneously to locate collisions. If the sets correspond to the evaluations of two functions h and g (*i.e.* $\mathcal{S} = \{h(x) : x\}$, $\mathcal{T} = \{g(y) : y\}$), we can use the memory-less algorithms by defining a helper function

$$\phi(x) = \begin{cases} h(x) & \text{if } x \text{ is odd} \\ g(x) & \text{if } x \text{ is even} \end{cases}$$

With constant probability, a collision in ϕ is a collision between h and g. For further details about algorithmic aspects of collision finding, we refer to the work of Joux [Jou09].

1.5.2 Birthday-bound security

Birthday attacks are very common across symmetric cryptography; they affect hash functions, MACs, block cipher modes of operation, and even stream ciphers [Gol97]. In most cases, there is a matching proof and the security of the construction is well understood. However, it is important to study the security loss around the birthday bound to better understand the modes, and to select adequate parameters for real-world use.

In particular, the security of a block cipher is often reduced to the key size κ: the best attack should be the exhaustive search of the key, with complexity 2^κ. However, birthday attacks on modes of operation show that the security is threatened when $2^{n/2}$ blocks of data are processed. Therefore, the block size n is also an important security parameter.

Surprisingly, this aspect of block cipher security is often overlooked, and block ciphers with small block size are commonly used ($n = 128$ is the typical case, but $n = 64$ is sometimes used for legacy or lightweight systems). This contrasts with the case of hash functions or stream cipher, where birthday attacks apply directly to the primitives and constructions with a state size n smaller than 160 have been deprecated.

1.5.3 Security beyond the birthday bound

In order to make block ciphers with a small block size more usable in practice, new modes of operation have been designed to stay secure after more than $2^{n/2}$ blocks of data. This requires very careful design, and advanced proof techniques to demonstrate security. Some of these modes are secure up $2^{2n/3}$, $2^{3n/4}$ or even 2^n blocks of data. For encryption, an important example is `CENC` [Iwa06] a modification of the counter mode secure up to roughly 2^n blocks of data [IMV16]. For authentication, several MACs have been proposed with security beyond the birthday bound, such as `SUM-ECBC` [Yas10] and `PMAC+` [Yas11]. Alternatively, several recent constructions use a tweakable block-cipher as the underlying primitive rather than a block cipher. This enables higher security without increasing the internal state size.

Outline

In this thesis, I present several results obtained by applying cryptanalysis techniques beyond primitives, to reveal properties of modes and protocols that are not captured by security proofs. Most of the results rely on finding collision between carefully crafted sets or functions, showing the importance of the birthday paradox across many aspects of symmetric cryptography.

In particular, I present many surprising or unexpected results:

- The sum of two hash functions ($h_1(m) \oplus h_2(m)$) is often *weaker* than the initial functions (Chapter 2);
- SHA-1 signatures can be abused in practice (Chapter 3);
- Secure channel protocols (`TLS`, `SSH`, `IPsec`) are broken by chosen-prefix collisions on the underlying hash function;
- The use of an internal checksum in a hash function (as in the GOST standards) makes it *weaker* when used in `HMAC` (Chapter 4);
- Block ciphers with a 64-bit block size (3DES, Blowfish) can be abused, leading to practical plaintext recovery (Chapter 5);
- The `CTR` mode leaks information about the plaintext after roughly $2^{n/2}$ blocks, just like the `CBC` mode (Chapter 5);
- Cryptanalysis techniques based on collisions can be used against several MACs with security beyond the birthday bound (Chapter 6);
- Widely used MAC constructions are broken in the quantum setting, even though quantum computers were believed to affect mainly public key cryptography (Chapter 7).

2. Generic Attacks Against Hash Functions and Hash Combiners

This chapter begins with a review of the state of the art, and includes new results based on articles [LW15] and [BDG+20], found in Appendix A and B.

Cryptographic hash functions are crucial components in many information security systems, used for various purposes such as building digital signature schemes, message authentication codes, commitments schemes, or password hashing functions. A hash function $H : \{0,1\}^* \mapsto \{0,1\}^n$ takes an arbitrary length input and produces an n-bit output or digest. Hash functions are used in many settings with various security requirements; the general expectation is that a hash function should behave like a random function from $\{0,1\}^*$ to $\{0,1\}^n$. More concretely, the main security notions expected from a hash function are:

Collision resistance. It should be hard to find two messages $M \neq M'$ with $H(M) = H(M')$.

Second-preimage resistance. Given a challenge message C, it should be hard to find $M \neq C$ with $H(M) = H(C)$.

Preimage resistance. Given a target hash value $\overline{H}$, it should be hard to find M with $H(M) = \overline{H}$.

Since generic collision attacks have complexity $2^{n/2}$, and generic preimage (and second-preimage) attacks have complexity 2^n, a secure hash function should have the same level of resistance.

Iterated hash functions

In practice, hash function are built by iterating a compression function with fixed-length inputs. In this chapter, we focus on narrow-pipe designs, *i.e.* we assume that the internal state size is the same as the output size n. The message M is first split into blocks $m_0, \ldots, m_{L-1}$; for simplicity, we assume that the message length is a multiple of the block size, and we ignore the padding. The hash function iterates a series of compression

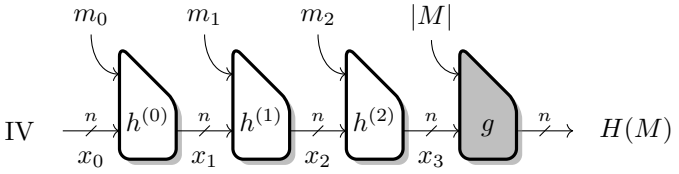


Figure 2.1: Iterated hash function.

functions $h^{(i)}$ over an internal state x, with the initial value denoted as IV. Finally, the hash value is computed with a finalization function g (see Figure 2.1):

$$x_0 = \text{IV} \qquad x_{i+1} = h^{(i)}(x_i, m_i) \qquad H(M) = g(x_L, |M|)$$

The finalization function g takes the message length as a parameter; this is denoted as MD strengthening. For simplicity, we sometimes write h instead of $h^{(i)}$, and $g(x)$ with the length being implicit.

This generic description includes two widely-used constructions as special case:

The Merkle-Damgård construction [Mer90; Dam90] uses the same compression function h for all steps ($\forall i, h^{(i)} = h$), as well as for the finalization ($g = h$). It used by standards SHA-1 [SHA1] and SHA-2 [SHA2].

The HAIFA construction [BD07] is a strengthened variant of the Merkle-Damgård construction where each compression function takes the block index as a parameter, and the finalization function takes a special parameter to make it different from the compression function. This avoids "cut-and-paste" attacks that move blocks of message around. This mode was used by several candidates in the SHA-3 competition, such as BLAKE [AHM+10] and Skein [FLS+10].

We use the notation h^* to denote the iterated compression function (without the finalization function): $h^*(m_0 \parallel m_1 \parallel m_2) = h_2(h_1(h_0(IV, m_0), m_1), m_2)$. For simplicity, we also abuse this notation with an extra input argument corresponding to the initial state: $h^*(x, m_0 \parallel m_1 \parallel m_2) = h(h(h(x, m_0), m_1), m_2)$ (with the block index implicit).

There has been a lot of work on generic attacks on iterated hash functions, in particular after the seminal multicollision attack of Joux [Jou04]. There are many security notions that fall between collision-resistance and preimage-resistance, and we need cryptanalysis to understand their security. In many cases, we find attacks with complexity close to the birthday bound, but some attacks are quite involved.

2.1 Generic attacks on hash functions

The Merkle-Damgård construction was proposed because it preserves collision resistance: any collision in the iterated hash function implies a collision in the compression function. In particular, if the compression function is collision-resistant (in the sense that no collision is known, and it is infeasible to find one explicitly), then the iterated hash function is collision-resistant. Generically, finding a collision in the compression function requires $2^{n/2}$ operations (due to the birthday paradox).

The Merkle-Damgård construction also preserves preimage resistance: a preimage in the iterated hash function implies a preimage of the finalization function. Generically, finding a preimage of the finalization function requires 2^n computations.

More generally, the Merkle-Damgård construction and its variants have been proven indifferentiable up to the birthday bound [CDM+05]. This means that the hash function behaves like a random oracle, assuming that the compression function is ideal.

However, several generic attacks have been proposed, showing that an attacker can break some security notions if she can perform more than $2^{n/2}$ operations. These attacks are based on special structures of messages, build from well-crafted collisions.

2.1.1 Collision extension

An important property of iterated hash function is that collisions in the internal state can be extended. Indeed, let us consider two messages M, M' of the same length. If $h^*(M) = h^*(M')$ then for any suffix S we have $h^*(M \parallel S) = h^*(M' \parallel S)$ (this implies $H(M \parallel S) = H(M' \parallel S)$).

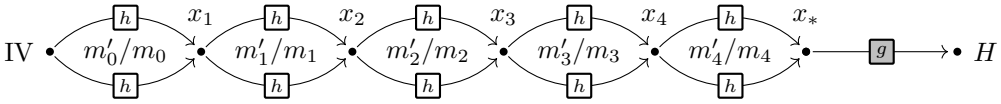


Figure 2.2: Joux's multicollision.

In particular, this distinguishes a narrow-pipe hash function from an ideal hash function using $\mathcal{O}(2^{n/2})$ operations (finding many related collisions is hard for an ideal hash function).

2.1.2 Multicollisions and attack against concatenation [Jou04]

Multicollision. A multicollision is a large set of messages $\mathcal{M}$ such that all messages have the same digest. In 2004, Joux introduced a simple technique to build a multicollision efficiently for narrow-pipe hash functions, as shown in Figure 2.2:

1. Starting from the IV, find a collision m_0, m'_0: $x_1 = h(\text{IV}, m_0) = h(\text{IV}, m'_0)$
2. Starting from $x_1 = h^*(m_0)$, find a collision m_1, m'_1: $x_2 = h(x_1, m_1) = h(x_1, m'_1)$
3. Starting from $x_2 = h^*(m_0 \,\|\, m_1)$, find a collision m_2, m'_2: $x_3 = h(x_2, m_2) = h(x_2, m'_2)$
4. ...

Since the internal states collide at every step, each collision is independent of the choice of the previous messages. After t iterations, we obtain a set of 2^t messages with the same digest: $[m_0 \,\|\, m_1 \,\|\, m_2 \,\|\, \ldots]$, $[m'_0 \,\|\, m_1 \,\|\, m_2 \,\|\, \ldots]$, $[m_0 \,\|\, m'_1 \,\|\, m_2 \,\|\, \ldots]$, $[m'_0 \,\|\, m'_1 \,\|\, m_2 \,\|\, \ldots]$, $[m_0 \,\|\, m_1 \,\|\, m'_2 \,\|\, \ldots]$, etc. Each step requires roughly $2^{n/2}$ work to find a collision, so that a multicollision of size 2^t is built with complexity $t \cdot 2^{n/2}$ in a narrow-pipe function. However, it would require roughly $2^{n \cdot (2^t - 1)/2^t}$ operations for an ideal n-bit hash function (almost 2^n for moderately large values of t).

Breaking the concatenation of two hash functions. Joux used multicollisions to break the concatenation of two hash functions: $x \mapsto H_1(x) \,\|\, H_2(x)$, with H_1 and H_2 two narrow-pipe hash functions, each with a state size of n bits. Surprisingly, the concatenation offers the same security as a single hash function, even though it has a larger internal state and a larger output size.

In order to find a collision, the attacker first builds a multicollision $\mathcal{M}$ of size $2^{n/2}$ for H_1. Then she evaluates H_2 over all messages in the multicollision, in order to find two messages $(M, M') \in \mathcal{M}$ with $H_2(M) = H_2(M')$. This produces a collision for the concatenation with complexity $\tilde{\mathcal{O}}(2^{n/2})$ because all messages in $\mathcal{M}$ have the same hash under H_1.

In order to find a preimage of a target value $\overline{H}_1 \,\|\, \overline{H}_2$, the attacker first builds a multicollision $\mathcal{M}$ of size 2^n for H_1 with final state x_*. Then she tries random blocks r until she finds one such that $g_1(h_1(x_*, r)) = \overline{H}_1$ (therefore, $\forall M \in \mathcal{M}, H_1(M \,\|\, r) = \overline{H}_1$). Finally, she iterates over all messages $M \in \mathcal{M}$, until $H_2(M \,\|\, r) = \overline{H}_2$. This produces a preimage for the concatenation with complexity $\tilde{\mathcal{O}}(2^n)$.

2.1.3 Expandable messages and second-preimage attack with a long challenge [KS05]

Expandable message. An expandable message is a large set of messages $\mathcal{M}$ of different lengths, such that all messages have the same digest. It is obtained with the same process as a multicollision, using messages pairs (m, m') of different lengths at each step (see Figure 2.3).

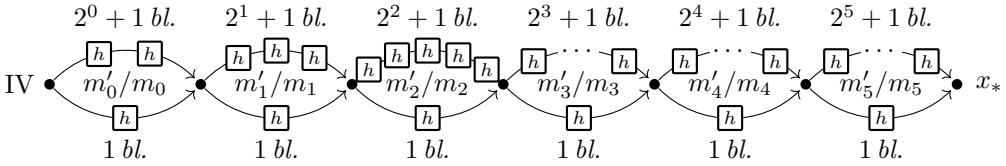


Figure 2.3: Kelsey and Schneier's expandable message.

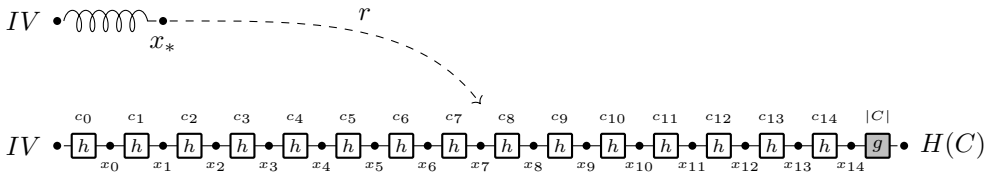


Figure 2.4: Second-preimage attack.

Second-preimage attack. The Merkle-Damgård construction resists preimage attacks, but Kelsey and Schneier have proposed a second-preimage attack with complexity $2^t + 2^{n-t}$, assuming that we are given a challenge message C of length 2^t (Figure 2.4).

The attacker first evaluates the hash function over the challenge message C, and stores all 2^t internal states x_i encountered during the computation of $H(C)$, so that connecting to a random x_i only requires 2^{n-t} random trials due to the birthday paradox. In order to deal with the MD strengthening, the attacker builds an expandable message $\mathcal{M}$ covering lengths up to 2^t, with final state x_*. Then she tries random blocks r until $h(x_*, r) = x_{i^*} \in \{x_i\}$. We denote the final part of the challenge starting from x_{i^*} as C_*. Finally, the attacker selects the message m of length $i^* - 1$ in the expandable message, and builds the second preimage as $m \parallel r \parallel C_*$. We have $h^*(m \parallel r) = x_{i^*}$, therefore $h^*(m \parallel r \parallel C_*) = h^*(C)$; moreover the new message has the same length as the challenge, so that the finalization function is the same, and $H(m \parallel r \parallel C_*) = H(C)$.

The attack is not applicable to constructions with different compression functions $h^{(i)}$ at each step. Actually, this was one of the motivations to introduce the HAIFA framework.

2.1.4 Diamond structures and herding attack [KK06]

Diamond structure. Given a set of internal states x_i, the *diamond structure* of Kelsey and Kohno is a set of messages S_i such that all internal states reach a common value x_*: $\forall i, h^*(x_i, S_i) = x_*$.

Kelsey and Kohno use an iterative procedure to reduce the size of the set $\{x_i\}$. Starting from 2^t internal states, they evaluate the compression function with $2^{n/2-t/2}$ random message blocks from each state. Each state is part of a collision with high probability: there are $2^{n/2-t/2}$ successors of a given state, and $2^{n/2+t/2}$ successors of the other states. They select a set of collisions without overlapping states, in order to obtain a new set of internal states for the next block with (essentially) half the size of the initial set. After $\mathcal{O}(t)$ iterations, a single final state is reached, so that building the full structure requires $\tilde{\mathcal{O}}(2^{(n+t)/2})$ operations. The structure has the shape of a binary tree (as shown by Figure 2.5), and the message S_i corresponding to an input state x_i is obtained by following the path from a leaf to the root.

Herding attack. Kelsey and Kohno use this structure to attack commitment schemes: an attacker can publish the hash value $g(x_*)$ (after the finalization function), and later build a message with this hash value using the diamond structure. This attack seems almost

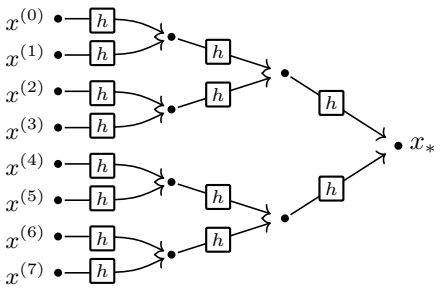


Figure 2.5: Kelsey and Kohno's diamond structure.

as hard as finding preimages, but it is in fact significantly easier for narrow-pipe hash functions. They describe two different scenarios:

- If the attacker has a small set of 2^t predefined messages M_i, she builds a diamond structure starting from the states obtained after processing the messages ($x_i = h^*(M_i)$) with complexity $2^{(n+t)/2}$. She publishes the common hash value $g(x_*)$, and she knows the messages S_i such that $H(M_i \parallel S_i) = g(x_*)$ for each i.

- If the attacker wants to herd any possible message to a fixed hash value, she builds a diamond structure from a set of $2^{n/3}$ arbitrary internal states, with complexity $2^{2n/3}$. After publishing the common hash value $g(x_*)$, she can generate a message S such that $H(C \parallel S) = g(x_*)$ for any challenge C with complexity $2^{2n/3}$. She just has to evaluate $h^*(C \parallel r)$ for $2^{2n/3}$ choices of r, until reaching one of the states x_i in the diamond structure. Then she recovers the message S_i from the diamond structure to build the full message $S = r \parallel S_i$ with $H(C \parallel S) = H(C \parallel r \parallel S_i) = g(x_*)$.

2.2 Combiners

In order to build systems that are more resilient, a natural idea is to combine several cryptographic primitives, so that a weakness in one primitive does not break the whole system. In particular, this has been used in TLS version 1.0 and 1.1, with the XOR of two MAC algorithms (`HMAC-MD5` and `HMAC-SHA-1`); the designers explain [RFC2246]: "In order to make the PRF as secure as possible, it uses two hash algorithms in a way which should guarantee its security if either algorithm remains secure."

There are two classical combiners for hash function: the concatenation combiner $H_1(M) \parallel H_2(M)$ and the XOR combiner $H_1(M) \oplus H_2(M)$ (Table 2.1). The concatenation combiner has the advantage of being *robust*: as long as one of the function is collision-resistant, the concatenation is collision-resistant. However, we have seen that it does not increase the security, because of the multicollision attack of Joux.

The security of the XOR combiner is not as well understood. Hoch and Shamir [HS08] proved that it is indifferentiable from a random oracle up to the birthday bound when the compression functions are modelled as random oracles (or even as *weak* random oracles). In particular, this implies security at least $2^{n/2}$ for collisions and preimages. For collision resistance, the bound is tight, since it is matched with the generic birthday attack bound. However, the security against preimage attacks is an open problem: there is a trivial attack with complexity 2^n, but this leaves a gap between the best attack and the known security proof.

Our results. In the following we describe new attacks against the XOR combiner, when using two narrow-pipe hash functions with state size n bits. We first describe a new

Table 2.1: Classical combiners with two n-bit narrow-pipe hash functions.

Combiner	Concatenation		XOR	
	$H_1(M) \,\|\|\, H_2(M)$		$H_1(M) \oplus H_2(M)$	
Robustness	Robust collision-resistance		Robust PRF	
Security (up to log factors)	Attack	Proof	Attack	Proof
Preimage security				
Ideal hash functions	2^{2n}	2^{2n}	2^n	2^n
HAIFA	2^n [Jou04]	2^n	$2^{5n/6}$ § 2.4	$2^{n/2}$ [HS08]
Merkle-Damgård	2^n [Jou04]	2^n	$2^{3n/5}$ § 2.6	$2^{n/2}$ [HS08]
Collision security				
Ideal hash functions	2^n	2^n	$2^{n/2}$	$2^{n/2}$
HAIFA	$2^{n/2}$ [Jou04]	$2^{n/2}$	$2^{n/2}$	$2^{n/2}$ [HS08]
Merkle-Damgård	$2^{n/2}$ [Jou04]	$2^{n/2}$	$2^{n/2}$	$2^{n/2}$ [HS08]

structure, the interchange structure in Section 2.3. Using this structure, we obtain an attack with complexity $2^{5n/6}$ using short messages, applicable to any narrow-pipe hash function, in Section 2.4. Then we describe a different attack using cycles in the functional graph, with complexity $2^{3n/4}$, in Section 2.5. This attack uses long messages and is only applicable to Merkle-Damgård hash functions. Finally, we present an improved attack with complexity $2^{3n/5}$ in Section 2.6, combining ideas from the two previous attacks.

2.3 The interchange structure

We now consider two narrow-pipe hash functions H_1 and H_2 with compression functions respectively $h_1^{(i)}$ and $h_2^{(i)}$, and finalization functions g_1 and g_2. The interchange structure is a new structure to control *simultaneously* the behavior of two independent hash computations which share the same input message. An interchange structure is a set of messages $\mathcal{M}$ and two sets of internal states, $\mathcal{A}$ for H_1 and $\mathcal{B}$ for H_2, such that: for any value A from $\mathcal{A}$ and any value B from $\mathcal{B}$, we can derive a message $M_{A,B}$ from $\mathcal{M}$ such that $h_1^*(M_{A,B}) = A$ and $h_2^*(M_{A,B}) = B$. Therefore, we can select states from $\mathcal{A}$ and $\mathcal{B}$ independently using this structure.

The main idea is to consider several chains of internal states reached by processing a common message $\boldsymbol{M}$ from different starting points (note that the message $\boldsymbol{M}$ is not fixed in advance, but will be determined when building the structure). More precisely, the message $\boldsymbol{M}$ is denoted as the *primary* message, and divided in several chunks: $\boldsymbol{M} = M_0 \| M_1 \| \dots$ (as discussed later, a chunk will consist of several message blocks). We denote chains of internal states for H_1 as a_j, and the individual states of the chain as a_j^i, with $h_1^*(a_j^i, M_i) = a_j^{i+1}$. Similarly, we denote chains for H_2 as b_k, with $h_2^*(b_k^i, M_i) = b_k^{i+1}$. When considering both hash functions, message block M_i leads from the pair of states (a_j^i, b_k^i) to (a_j^{i+1}, b_k^{i+1}), which is denoted:

$$(a_j^i, b_k^i) \overset{M_i}{\rightsquigarrow} (a_j^{i+1}, b_k^{i+1}).$$

2.3.1 Switch structure.

First we build special structures called *switches* in order to jump between chains in a controlled way. A switch allows jumping from a specific pair of chains (a_{j_0}, b_{k_0}) to a different pair of chains (a_{j_0}, b_{k_1}) using a secondary message chunk M_i', in addition to the

normal transitions using chunk M_i of the primary message $\boldsymbol{M}$:

$$(a_j^i, b_k^i) \overset{M_i}{\rightsquigarrow} (a_j^{i+1}, b_k^{i+1}) : \qquad \text{normal transition for each chain } (\forall j \forall k)$$
$$(a_{j_0}^i, b_{k_0}^i) \overset{M_i'}{\rightsquigarrow} (a_{j_0}^{i+1}, b_{k_1}^{i+1}) : \qquad \text{jump from chains } (a_{j_0}, b_{k_0}) \text{ to } (a_{j_0}, b_{k_1})$$

The main message chunk M_i and the secondary message chunk M_i' are determined when building the switch, with the following constraints:

$$h_1^*(a_{j_0}^i, M_i) = h_1^*(a_{j_0}^i, M_i') \qquad\qquad h_2^*(b_{k_1}^i, M_i) = h_2^*(b_{k_0}^i, M_i')$$

We start with a multicollision of size $2^{n/2}$ satisfying the first equality, and select two messages (M_i, M_i') satisfying the second equality. The chains are defined by following the main message:

$$a_j^{i+1} = h_1^*(a_j^i, M_i) \qquad\qquad b_k^{i+1} = h_2^*(b_k^i, M_i)$$

This defines a switch structure with complexity $\tilde{\mathcal{O}}(2^{n/2})$. A simple example is depicted in Figure 2.6. We note that the secondary message chunk M_i' should only be used when the state is $(a_{j_0}^i, b_{k_0}^i)$.

2.3.2 Interchange structure.

By combining several switches, we build an interchange structure with starting points IV_1 and IV_2 and ending points $\{A_j\}$ and $\{B_k\}$ $(0 \leq j, k < 2^t)$, so that we can select a message ending in any state (A_j, B_k). Figure 2.7 shows one possible way to build such a structure, and Figure 2.8 shows how to select a given message in the structure. The chains $a_j, j \neq 0$ and $b_k, k \neq 0$ are initialized arbitrarily, and the message chunks are determined when building the switches.

An interchange structure with 2^t chains for each function requires about 2^{2t} switches. Since we can build a switch for a cost of $\tilde{\mathcal{O}}(2^{n/2})$, the total structure is built with $\tilde{\mathcal{O}}(2^{2t+n/2})$ operations.

2.4 Preimage attack against the XOR combiner

Finally, we use an interchange structure with ending points $\{A_j\}$ and $\{B_k\}$, $(0 \leq j, k < 2^t)$ to build a preimage attack as follows. Let $\overline{H}$ denote the target value. We select a random message block r, and we compute two lists by evaluating the compression functions after the interchange structure: $\{A_j' = g_1(h_1(A_j, r))\}$ and $\{B_k' = \overline{H} \oplus g_2(h_2(B_k, r))\}$. We expect a match between the lists with probability 2^{2t-n}. After about 2^{n-2t} random choices of r, we get a match (j^*, k^*): $g_1(h_1(A_{j^*}, r)) = \overline{H} \oplus g_2(h_2(B_{k^*}, r))$. Therefore, we construct a preimage of $\overline{H}$ by concatenating the message leading to (A_{j^*}, B_{k^*}) in the interchange structure, and the block r. We obtain

$$H_1(M_{A_{j^*}, B_{k^*}} \,\|\, r) \oplus H_2(M_{A_{j^*}, B_{k^*}} \,\|\, r) = g_1(h_1(A_{j^*}, r)) \oplus g_2(h_2(B_{k^*}, r)) = \overline{H}$$

Complexity analysis. Building the interchange structures requires $\tilde{\mathcal{O}}(2^{2t+n/2})$ evaluations of the compression function, while the preimage search requires about 2^{n-t} evaluations. The optimal complexity is reached when both steps take the same time, *i.e.* $t = n/6$. This gives a complexity of $\tilde{\mathcal{O}}(2^{5n/6})$.

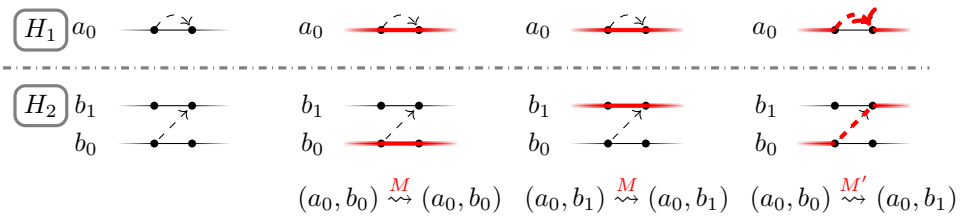


Figure 2.6: A single switch: jump from (a_0, b_0) to (a_0, b_1) by using M' (dashed lines) instead of M (solid lines).

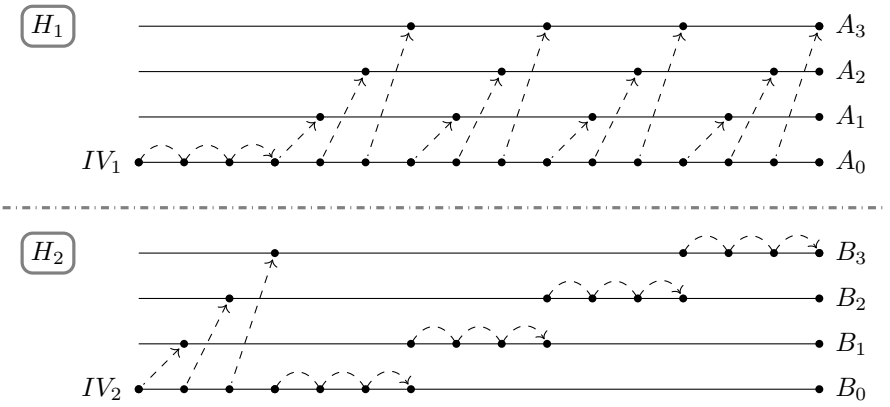


Figure 2.7: Overview of an interchange structure.

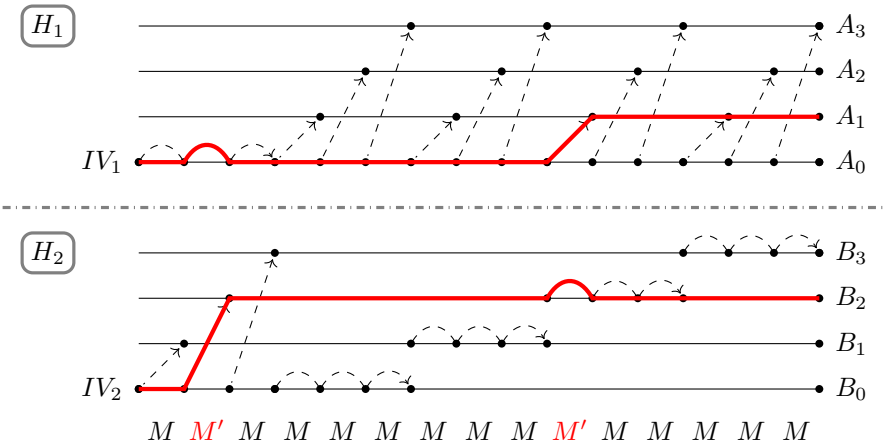


Figure 2.8: Using of the interchange structure to reach output (A_1, B_2).

2.5 Preimage attack using cycles in the functional graph

We now assume that the hash functions H_1 and H_2 use the Merkle-Damgård construction, so that the compression function is the same at every step. This gives more freedom to the attacker, and more advanced techniques have been proposed to find preimages of the XOR combiner with a smaller complexity. In the section, we present an alternative description of the multi-cycle idea introduced in [BWG+17], to obtain a simplified attack with complexity $\tilde{\mathcal{O}}(2^{3n/4})$.

This attack uses long messages that repeat a fixed message block β (for instance, $\beta = 0$). When evaluating h^* on such a message, we iterate a fixed function $x \mapsto h(x, \beta)$. Assuming that this function behaves as a random function, we expect to reach a cycle after roughly $2^{n/2}$ iterations, and the cycle has length roughly $2^{n/2}$. More precisely, there are several components in the functional graph with cycles of different lengths; the main component has expected size 0.76×2^n with a cycle of expected size $\sqrt{\pi/8} \times 2^{n/2}$ [FO90].

We iterate the function $h_1(\cdot, \beta)$ starting from IV_1, until we reach a cycle; we denote the cyclic points as $\{A_j\}$ and its length as ℓ_1. Similarly, we obtain a cycle $\{B_k\}$ of length ℓ_2 for the function $h_2(\cdot, \beta)$ starting from IV_2. We assume that ℓ_1 and ℓ_2 are relatively prime[1] with $\ell_1 < \ell_2$, and of the order of $2^{n/2}$.

This simple structure provides an alternative way to control independently the behavior of H_1 and H_2, by varying the length of a message $M = \beta^\lambda$. Indeed, with $\lambda > \mu$, with $\mu = \Theta(2^{n/2})$ the distance to reach both cycles, the final state is of form (A_j, B_k); moreover, with a suitable choice of the indices of A_j and B_k we have:

$$h_1^*(\beta^\lambda) = A_{\lambda \bmod \ell_1} \qquad\qquad h_2^*(\beta^\lambda) = B_{\lambda \bmod \ell_2}$$

Therefore, we obtain a message reaching a specific state (A_{j^*}, B_{k^*}) by solving a set of modular equations: $\lambda \equiv j^* \bmod \ell_1$, $\lambda \equiv k^* \bmod \ell_2$. Using the Chinese remainder theorem, we deduce a length $\lambda \geq \mu$ such that $h_1^*(\beta^\lambda) = A_{j^*}$ and $h_2^*(\beta^\lambda) = B_{k^*}$. For a random (j^*, k^*), the length λ is uniformly distributed between μ and $\mu + \ell_1\ell_2$.

The full attack requires an expandable message to deal with the MD strengthening. It produces a message of length $2^{3n/4}$ with the following steps, as shown in Figure 2.9:

1. Build an expandable message $\mathcal{M}$, with maximum length $2^{3n/4}$, with final states s_1, s_2.

2. Find the cycle $\{A_j\}$ of $h_1(\cdot, \beta)$ and the cycle $\{B_k\}$ of $h_2(\cdot, \beta)$, starting from s_1 and s_2.

3. For a random block r, match $\{g_1(h_1(A_j, r))\}$ and $\{g_2(h_2(B_k, r)) \oplus \overline{H}\}$.

   If there is a match (j^*, k^*), find the length λ such that $h_1^*(s_1, \beta^\lambda) = A_{j^*}$ and $h_2^*(s_2, \beta^\lambda) = B_{k^*}$ using the Chinese remainder theorem.

   If $\lambda < 2^{3n/4}$, select the message m of length $2^{3n/4} - \lambda - 1$ in the expandable message. The preimage is $M = m \parallel \beta^\lambda \parallel r$.

   Otherwise, repeat Step 3.

Complexity analysis. Step 1 has complexity $\mathcal{O}(2^{3n/4})$ and step 2 has complexity $\mathcal{O}(2^{n/2})$.

Each iteration of Step 3 (for each try of r) has complexity $\mathcal{O}(2^{n/2})$. On average there is a single match (j^*, k^*), and the probability that $\lambda < 2^{3n/4}$ is $2^{-n/4}$. Therefore, we expect $2^{n/4}$ iterations, with a total complexity $2^{3n/4}$.

[1] This happens with probability $6/\pi^2 \approx 0.61$ for random integers.

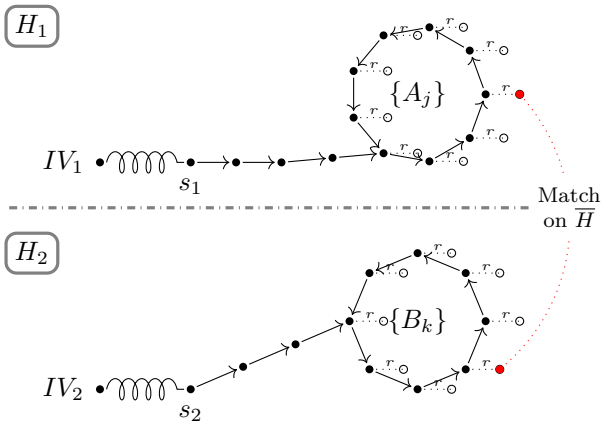


Figure 2.9: Preimage attack on the XOR combiner based on cycles.

2.6 Improved preimage attack using cycles in the functional graph

Finally, we present the best known attack, with complexity $2^{3n/5}$. This unpublished attack is an improved variant of the attack given in [BDG+20], using the short cycle idea from [GHK+23]; it is a joined work with Xavier Bonnetain, Rachel Heim Boissier and André Schrottenloher.

The first improvement is to consider several connections from the IV to the cycle. In the final step, each target (A_j, B_k) can now be reached in many different ways, and this improves the probability that one of the corresponding messages β^λ is short. In order to build the connections efficiently, we precompute $2^{3n/5}$ points in the functional graph, together with their distance to a fixed cyclic point. Therefore, starting from a random point, we only need $2^{2n/5}$ iteration to reach a precomputed point, and to deduce the connection to the cycle. Moreover, we use an interchange structure to provide independent choices of the connections in H_1 and H_2.

The second improvement is to select a value β such that the cycles in the main component of $h_1(\cdot, \beta)$ and $h_2(\cdot, \beta)$ are shorter than average, with $\ell_1, \ell_2 \lessapprox 2^{9n/20}$. This requires on average $2^{n/10}$ random choices of β, and each trial requires $2^{n/2}$ evaluations of the compression function in order to identify the main cycle. Thanks to the shorter cycles, the length λ computed from the Chinese remainder theorem will be at most $\ell_1\ell_2 \approx 2^{9n/10}$.

The attack produces a message of length $2^{3n/5}$ with the following steps (see Figure 2.10):

1. Build an expandable message $\mathcal{M}$, with maximum length $2^{3n/5}$, with final state s_1, s_2.
2. Build an interchange structure $\mathcal{I}$ with $2^{n/20}$ endpoints $\{x_u\}$ and $\{y_v\}$, starting from (s_1, s_2).
3. Find β such that $\ell_1, \ell_2 \lessapprox 2^{9n/20}$. Denote the cycle of $h_1(\cdot, \beta)$ as $\{A_j\}$ and the cycle of $h_2(\cdot, \beta)$ as $\{B_k\}$.
4. Precompute a set of $2^{3n/5}$ points in the main component of $h_1(\cdot, \beta)$ and $h_2(\cdot, \beta)$, with known distances to the main tree root.
5. Choose $2^{3n/20}$ random blocks ρ_w; for all interchange endpoints x_u and y_v compute the distance from $h_1(x_u, \rho_w)$ and $h_2(y_v, \rho_w)$ to the respective cycle.
6. For a random block r, match $\{g_1(h_1(A_j, r))\}$ and $\{g_2(h_2(B_k, r)) \oplus \overline{H}\}$.

   If there is a match (j^*, k^*), then for each x_u, y_v, ρ_w, find the length λ such that $h_1^*(x_u, \rho_w \parallel \beta^\lambda) = A_{j^*}$ and $h_2^*(y_v, \rho_w \parallel \beta^\lambda) = B_{k^*}$ using the Chinese remainder theorem.

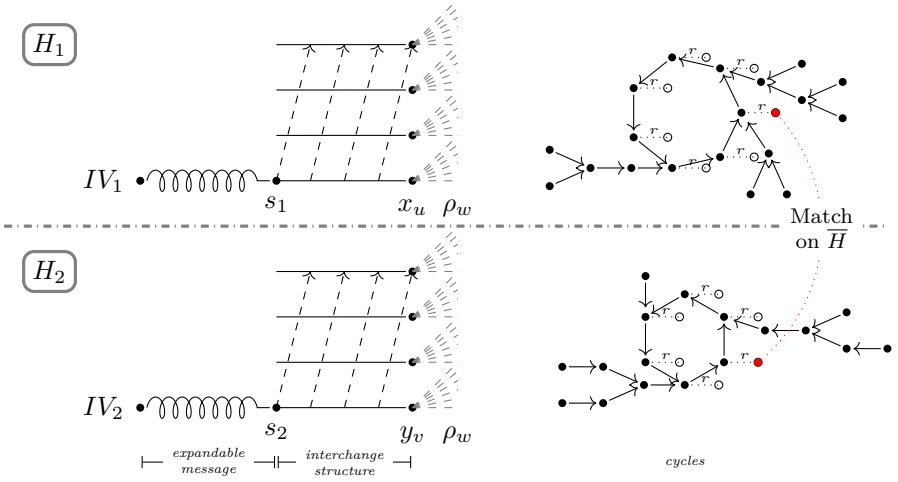


Figure 2.10: Improved preimage attack on the XOR combiner based on cycles.

If $\lambda < 2^{3n/5}$, select the message μ such that $h_1^*(s_1, \mu) = x_u, h_2^*(s_2, \mu) = y_v$ in the interchange structure and the message m of length $2^{3n/5} - \lambda - 2 - |\mu|$ in the expandable message. The preimage is $M = m \parallel \mu \parallel \rho_w \parallel \beta^\lambda \parallel r$.

Otherwise, repeat Step 6.

Complexity analysis. Step 1 has complexity $\mathcal{O}(2^{3n/5})$; step 2 has complexity $\tilde{\mathcal{O}}(2^{n/2} + 2^{2\times n/20}) = \tilde{\mathcal{O}}(2^{3n/5})$; step 3 has complexity $\tilde{\mathcal{O}}(2^{3n/5})$; step 4 has complexity $\mathcal{O}(2^{3n/5})$.

Step 5 has complexity $\mathcal{O}(2^{3n/20} \times 2^{n/20} \times 2^{2n/5}) = \mathcal{O}(2^{3n/5})$ using the points precomputed at step 4. Since we use the main component of $h_1(\cdot, \beta)$ and $h_2(\cdot, \beta)$, each point $h_1(x_u, \rho_w)$ or $h_2(y_v, \rho_w)$ has a constant probability to reach the cycle.

Each iteration of Step 6 (for each try of r) has complexity $\ell_2 = \mathcal{O}(2^{9n/20})$. There is a match (j^*, k^*) with probability $\mathcal{O}(2^{2\times 9n/20 - n}) = \mathcal{O}(2^{-n/10})$. In this case we obtain $2^{n/20} \times 2^{n/20} \times 2^{3n/20} = 2^{n/4}$ candidates λ; each one is smaller than $2^{3n/5}$ with probability $\Omega(2^{3n/5}/2^{2\times 9n/20}) = \Omega(2^{-3n/10})$; in total, we obtain one $\lambda < 2^{3n/5}$ with probability $2^{n/4} \times 2^{-3n/10} = 2^{-n/20}$. Therefore, we expect $\mathcal{O}(2^{3n/20})$ iterations of Step 6, with a total complexity $\mathcal{O}(2^{3n/20} \times 2^{9n/20}) = \mathcal{O}(2^{3n/5})$.

2.7 Conclusion

The interchange structure is a novel construction that gave the first generic attack against the XOR combiner. This result is rather surprising: the sum of two ideal narrow-pipe hash functions only has about $5n/6$ bits of security against preimage attacks. In particular, the *sum is weaker* than the initial functions. More advanced attacks reduce this complexity in the case of Merkle-Damgård hash functions: the best known attack has complexity $2^{3n/5}$. Since many practical hash functions are narrow-pipe (*e.g.* SHA-1, SHA-2, Whirlpool, RIPEMD, GOST, BLAKE, Skein...), the XOR combiner will usually provide a weaker security than the component hash functions.

In [BDG+20] (Appendix B), we also propose new attacks against more advanced combiners such as the "hash twice" construction, and the Zipper hash. These attacks benefit from the use of the interchange structure.

3. Chosen-Prefix Collision Attacks

This chapter is based on articles [LP19], [LP20] and [BL16b], found in Appendix C, D and E.

The main security property expected from cryptographic hash functions is *collision resistance*: it should be hard for an adversary to compute two distinct messages M and M' that map to the same hash value $H(M) = H(M')$. While many cryptographic constructions rely on collision-resistance for their security proofs, collision attacks are hard to turn into a break of concrete protocols, because the adversary has limited control over the colliding messages.

Impact of collision attacks. Collision attacks return random-looking messages M and M'. In order to break a real-world application, they must somehow be hidden as junk inside meaningful messages.

An attacker can exploit the collision extension property: she chooses a common prefix P before computing the collision (M, M'), and adds a common suffix S after the collision. This results in colliding messages $H(P \| M \| S) = H(P \| M' \| S)$ where P and S are chosen meaningful messages. This technique was used to mount a "poisoned-message" attack against document formats with some level of programmability [DL05; GIS06; Alb17]. The attacker builds a prefix that hides the collision blocks by defining a variable, and uses an `if-then-else` switch depending on that variable to display two different contents. She obtains two messages with the same hash that display a different meaningful content, depending on which message of the colliding pair (M, M') is inserted. As a concrete example, the SHA-1 collision of Stevens *et al.* [SBK+17] used this trick to produce two colliding PDFs.

However, this structure can be detected in the resulting document. More generally, this approach is usually not sufficient to break real-world applications.

Chosen-prefix collision attacks. A *chosen-prefix* collision attack [SLW07] is a stronger variant of collision attack. The attacker is challenged with two message prefixes P and P', and her goal is to compute two messages M and M' such that $H(P \| M) = H(P' \| M')$. Chosen-prefix (CP) collisions are usually harder to produce than (identical-prefix) collisions, but the practical impact of such an attack is more significant. A CP collision attack indicates the ability of an attacker to obtain a collision even though random differences (thus potentially some meaningful information) were inserted as message prefix.

Application to digital certificates. A digital certificate is essentially a signature of an identity and a cryptographic key: $C = \mathsf{sign}(\mathsf{id}, k)$. The user presents her identity information and cryptographic key to a certificate authority who computes the signature. In practice, the signature is computed over a hash: $C = \mathsf{sign}(H(\mathsf{id} \| k))$. Therefore, if an attacker generates a collision $H(\mathsf{id} \| k) = H(\mathsf{id}' \| k')$, she can request a certificate for (id, k) (assuming the identity id is correct) and reuse the signature as a certificate for (id', k').

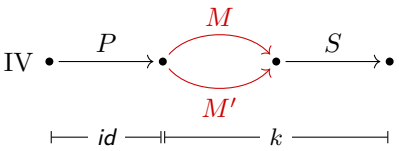


Figure 3.1: Colliding certificates using an identical-prefix collision.

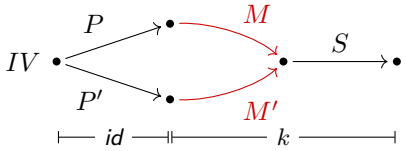


Figure 3.2: Colliding certificates using a chosen-prefix collision.

Using an identical-prefix collision attack, it is hard to generate a collision with different meaningful identities. However, it is possible to generate a collision with two different keys for the same identity. The attacker starts the collision attack after processing id, and hides the colliding messages M, M' inside the cryptographic keys (Figure 3.1). This was demonstrated on X.509 certificated signed with MD5 [LW05]. The attacker generates RSA moduli with known factorization by choosing only the low-order bits.

A chosen-prefix collision gives more freedom to the attacker. She can use two different identities $\mathsf{id}, \mathsf{id}'$ as prefixes, and compute a collision (Figure 3.2). If id is her identity, and id' the identity of a victim, this results in an impersonation attack, because she obtains a valid certificate under the name of id'. This was demonstrated on X.509 certificates signed with MD5 [SLW07], and a more advanced variant of this attack resulted in a rogue certificate authority [SSA+09].

Our results. In this chapter we present new chosen-prefix collision attacks, and new applications of CP collisions. In particular, we have computed the first SHA-1 CP collision. The main goal of those results is to demonstrate that weaknesses of MD5 and SHA-1 are not just theoretical issues: they can be exploited to mount practical attacks against real-world systems.

We assume that the hash function considered is an n-bit narrow pipe primitive, based on a Merkle-Damgård-like operating mode. In addition, we assume that the compression function is built upon a block cipher in Davies-Meyer mode: $h(x, m) = E_m(x) + x$, with $+$ a group operation (MD5 and SHA-1 use modular addition).

3.1 From collisions to chosen-prefix collisions

The most widely used hash functions since the 1990's are based on similar design principles, and are collectively called the MD-SHA family: MD4 [Riv91], MD5 [RFC1321], SHA-1 [SHA1], SHA-2 [SHA2] (SHA-2 is a family of functions including SHA-224, SHA-256, SHA-384 and SHA-512). There is a large body of previous work on cryptanalysis of the MD-SHA family. In particular, the early members of the family have been broken in practice, with actual collisions given for MD4 [Dob96; WLF+05], MD5 [WY05], and SHA-1 [WYY05; SBK+17] (Table 3.1). On the other hand, SHA-2 is still considered secure. SHA-3 does not follow these design principles and is also considered secure.

3.1.1 Cryptanalysis of the MD-SHA family

Collision attacks against MD-SHA hash functions are based on differential cryptanalysis, where an attacker manages to somewhat control the output of the compression function. Several important ideas were used to turn differential cryptanalysis into an effective tool against hash functions:

Local collisions [Dob97]. An important trick to build high probability differential trails is to construct local collisions: small message differences whose influence is immediately corrected with other message differences inserted in the subsequent steps. In MD4

and MD5, this trick ensures many rounds without any difference in the internal state, so that those rounds don't contribute to the probability.

Linearization [CJ98]. In order to build differential trails with high probability for SHA-1, a linearized version of the step function is used. Thanks to the linearity of the message expansion, a combination of disturbances and corrections gives a series of local collisions. Differential trails with a low-weight output difference δ_O can be used to find near-collisions in the compression function (i.e. two outputs that are close to a collision).

Message modification [BC04; WYY05]. In a differential attack against a hash function, the attacker can choose messages that directly satisfy some of the constraints of the path, because there is no secret key. While the conditions in the first steps are easy to satisfy, more advanced techniques have been introduced to extend the usage of these degrees of freedom to later rounds in order to speed-up collision search: neutral bits (firstly introduced for cryptanalysis of SHA-0 [BC04; BCJ+05]), message modifications [WYY05] and boomerangs/tunnels [Kli06; JP07].

Non-linear trails [WYY05]. In order to get more flexibility on the differential trails, the first few steps can use non-linearity instead of following the constraints of the linearized step function. This does not affect the complexity of the search for conforming messages (thanks to message modification techniques), but it allows building trails from an arbitrary input difference to a good fixed output difference δ_O (or its opposite $-\delta_O$).

Multi-block technique [CJ98; WYY05]. The multi-block technique takes advantage of the Davies-Meyer construction used inside the compression function ($h(x, m) = E_m(x) + x$). Because of the feed-forward, an attacker can use several differential trails in E, and several near-collisions blocks, to iteratively affect the internal state. In particular, using non-linearity in the first steps, she can derive two related trails $0 \overset{\delta_M}{\rightsquigarrow} \delta_O$ and $\delta_O \overset{-\delta_M}{\rightsquigarrow} -\delta_O$ in E from a single linear trail, by swapping the message pair. When conforming messages are found for each block, this leads to a collision because the internal state differences cancel out (see Figure 3.3).

Birthday phase for chosen-prefix collisions [SLW07; Ste13]. Differential techniques have also been used for chosen-prefix collision attacks. An attacker can relax the last steps of the differential trail to allow a set $\mathcal{D}$ of output differences rather than a single δ_O. She can also use several differential trails, and use the union of the corresponding sets. Starting from two different prefixes P, P', the chosen-prefix collision attack has two stages (see Figure 3.4):

- In the *birthday stage*, the attacker uses a generic collision search with message blocks m_0, m'_0 to reach a difference $\delta = H(P' \parallel m'_0) - H(P \parallel m_0)$ in $\mathcal{D}$ with complexity roughly $\sqrt{\pi \cdot 2^n / |\mathcal{D}|}$.
- In the *near-collision stage*, she builds a differential trail $\delta \rightsquigarrow -\delta$ using non-linearity in the first steps, and searches for a conforming message to build the chosen-prefix collision.

Multi-block for chosen-prefix collisions [SLW07]. If a collection of differential trails affecting separate parts of the internal state is available, chosen-prefix collision attacks can be greatly improved. In particular, if an arbitrary input difference δ_R can be decomposed as $\delta_R = -\left(\delta_O^{(1)} + \delta_O^{(2)} + \cdots + \delta_O^{(r)}\right)$, where each $\delta_O^{(i)}$ can be reached as the output of a differential trail, the attacker just has to find near-collision blocks with output differences $\delta_O^{(1)}, \ldots, \delta_O^{(r)}$ (see Figure 3.5).

Alternatively, if this only covers a subset of input differences, the multi-block technique is combined with a birthday stage. This approach was notably applied to MD5 [SLW07].

3.1.2 New techniques for chosen-prefix collisions

The CP collision attack against MD5 [SLW07] uses dedicated differential paths in order to control separate parts of the internal state difference. This requires dedicated analysis to find suitable trails. In this section we present a more generic way to build CP collision attacks, reusing an existing collision attacks. This is particularly well-suited to SHA-1, given the limited freedom in the construction of differential trails.

The main improvement comes from how we use multiple near-collision blocks. For instance, let us consider a trail allowing a set $\mathcal{D}$ of output differences. Using two near-collision blocks we can start from any difference in the set $\mathcal{S} := \{\delta_1 + \delta_2 : \delta_1, \delta_2 \in \mathcal{D}\}$, and cancel it iteratively with a first block following a trail $\delta_1 + \delta_2 \rightsquigarrow -\delta_1$ and a second block following a trail $\delta_2 \rightsquigarrow -\delta_2$. The set $\mathcal{S}$ grows with the number of blocks: this reduces the cost of the birthday search in exchange for a more expensive near-collision stage.

While there are previous chosen-prefix collision attacks using several near-collision blocks [LPR+07; Pey07; SLW07; SSA+09; MRS15], these attacks use a collection of differential trails to impact different parts of the state (each block uses a different trail). Previous chosen-prefix collision attacks based on a single trail (against SHA-1 [Ste13] and MD5 [SSA+09, Section 6]) used only one near-collision block. On the opposite, our technique can be used with a single differential trail, or a collection of trails without any special property.

Overview of the attack. Our strategy is composed of two phases. After processing the two prefixes, the first phase uses a generic birthday search to reach a chaining value difference in a predefined set $\mathcal{S}$. In the second phase, we use several near-collision blocks to map the chaining variable difference to a collision, using techniques developed for collision attacks.

The second phase is the most challenging. The attacker uses a graph $\mathcal{G}$, where each node represents a chaining variable difference in the set $\mathcal{S}$. A directed edge from node i to node j represents a way for an attacker to reach chaining variable difference j from difference i with a single message block; the details of the differential trail are attached to the edge. A path in the graph from a node δ to the node 0 corresponds to a sequence of output differences that ends up with a collision.

3.1.3 Building the set $\mathcal{S}$ and the graph $\mathcal{G}$

We consider that the attacker knows some good core differential trails for the internal block cipher E, that is differential trails that go from early steps to late steps of E. We denote the set of all possible output differences as $\mathcal{D}$ (in particular, we have $0 \in \mathcal{D}$, and $\delta \in \mathcal{D} \iff -\delta \in \mathcal{D}$ because of symmetries).

We further assume that any input difference for E can be mapped to any of the core differential trails inside the primitive. With functions of the MD-SHA family, this is achieved with the non-linear part of the trail in the first steps of the function.

Simple graph. For simplicity, we fix a maximum number of block t for the near-collision phase. We deduce that the set of differences that can be corrected is

$$\mathcal{S} := \{\delta_1 + \delta_2 + \cdots \delta_t : \delta_1, \delta_2, \ldots \delta_t \in \mathcal{D}\}.$$

We build the graph $\mathcal{G}$ by adding an edge between nodes i and j when $i - j \in \mathcal{D}$. We obtain a simple attack by taking the shortest paths in this graph, as illustrated by Figure 3.6.

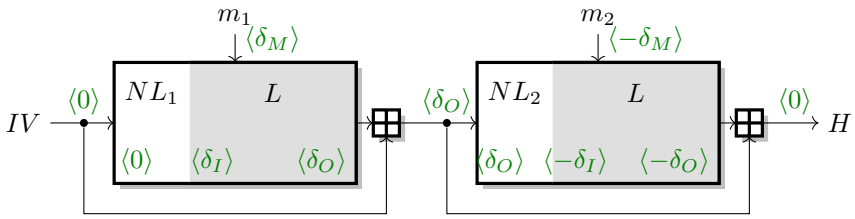


Figure 3.3: 2-block collision attack using a linear trail $\delta_I \overset{\delta_M}{\rightsquigarrow} \delta_O$ and two non-linear trails $0 \rightsquigarrow \delta_I$ and $\delta_O \rightsquigarrow -\delta_I$. Green values between bracket represent differences in the state.

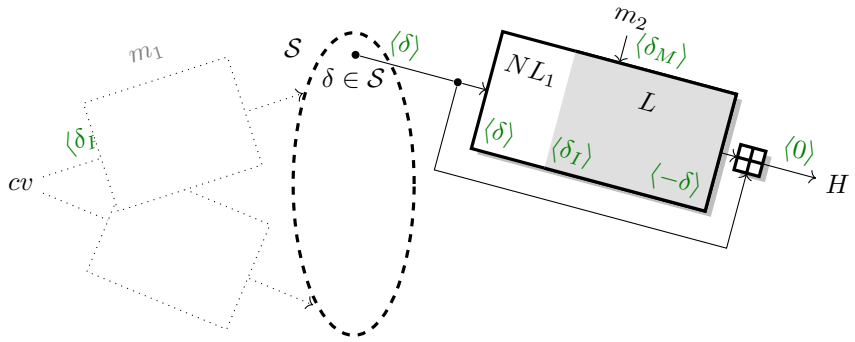


Figure 3.4: Single-block chosen-prefix collision attack with a birthday stage. The linear trail $\delta_I \rightsquigarrow \delta_O$ is relaxed to reach a set $\mathcal{S}$ of feasible differences.

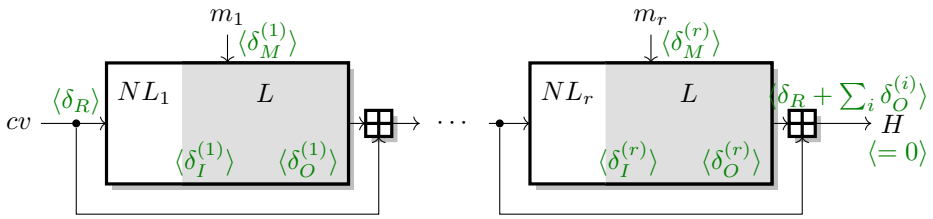


Figure 3.5: Multi-block chosen-prefix collision attack. We assume that an arbitrary difference δ_R can be decomposed as $\delta_R = -\left(\delta_O^{(1)} + \delta_O^{(2)} + \cdots + \delta_O^{(r)}\right)$, where each $\delta_O^{(i)}$ can be reached as the output of a differential trail.

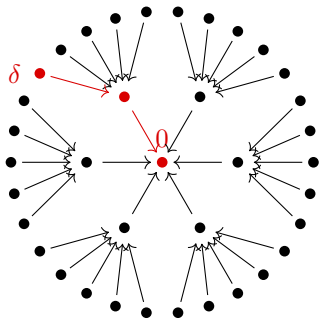


Figure 3.6: Graph search.

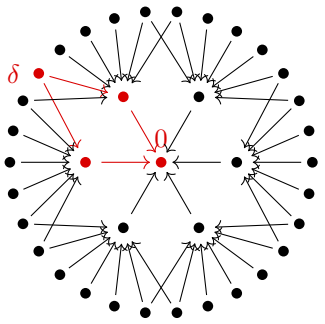


Figure 3.7: With clustering.

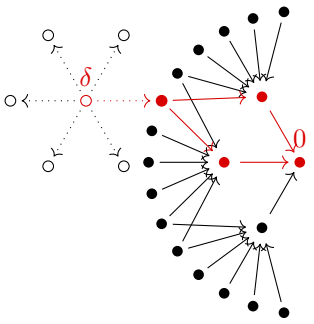


Figure 3.8: Bi-directional.

Clustering. In order to minimize the complexity of the near-collision phase of the attack, we use a clustering technique to exploit multiple paths in the graph (see Figure 3.7). Indeed, the search for near-collision blocks is a random process that generates many message pairs following the trail until some intermediate step, and verifies the final difference. Therefore, the attacker does not have to commit to a fixed output difference. If a single core trail produces several output differences that correspond to useful edges in the graph, the attacker can run the search for near-collision blocks and stop as soon as one of them is obtained.

Concretely, let us assume we have two output differences δ_1 and δ_2 compatible with the same differential trail, that can each be reached with a cost of C_{block}. There are two different ways to erase a difference $-\delta_1 - \delta_2$ in the state:

- a block with difference δ_1, followed by a block with difference δ_2;
- a block with difference δ_2, followed by a block with difference δ_1.

If we don't decide in advance the target difference for the first block, the search is expected to reach either δ_1 or δ_2 with a cost of only 0.5 C_{block}, leading to an attack complexity of 1.5 C_{block} rather than 2 C_{block}.

Bi-directional search. Since the CP collision attack is essentially a path search in a graph, we use a bi-directional search to reduce the size of the precomputed graph. More precisely, when we evaluate the cost of a node, instead of just looking it up in the graph, we recompute all edges starting from the current node to see if they reach the graph. Indeed, the complexity of the birthday search is essentially $\sqrt{\pi \cdot 2^n/|\mathcal{S}|}$, therefore we need a large set $\mathcal{S}$, but storing it becomes an issue.

This is illustrated by Figure 3.8, where black dots correspond to precomputed nodes stored in the graph, and white dots are only computed during the online phase. We can't compute exactly the size of the implicit graph, but we can evaluate it experimentally.

Implicit edges. We build the graph with a fixed set $\mathcal{D}$ of potential output differences with high probability. However, the messages produced by the near-collision block search can reach many other output differences. Some of them can be useful in our attack, even if they have a lower probability: we can use a block as long as the new state difference is closer to a collision. Therefore, during the near-collision phase, we also keep blocks that don't correspond to a difference in $\mathcal{D}$, and look up the new state difference in the graph.

3.2 Chosen-prefix collision attack against SHA-1

To apply the method to SHA-1, we reuse as much as possible the details of the collision attack from [SBK+17]. In particular, we use the best core trail known for attacks against SHA-1, and we reuse the GPU code from the collision attack [SBK+17]. We denote the complexity to find a block conforming to the trail (with an optimal output difference) as C_{block}; this cost was estimated as $C_{\text{block}} = 2^{64.7}$ SHA-1 evaluations on a GTX-970 GPU.

3.2.1 Improvements to near-collision block search

As a first contribution, we propose improvements to the collision attack from Eurocrypt 2013 [Ste13] and its GPU implementation from Crypto 2017 [SBK+17]. Through better use of degrees of freedom (message modifications and boomerangs) and code improvements, we gained a factor between 8 and 10 (depending on GPU architecture) on the time needed to find a conforming block.

With these improvements, we obtain the cost of a near-collision block as $C_{\text{block}} = 2^{61.2}$ SHA-1 evaluations on a GTX-970 GPU, and $C_{\text{block}} = 2^{61.6}$ on a GTX 1060.

3.2.2 Parameters of the attack

Differential trails. As in the previous chosen-prefix collision attack on SHA-1 [Ste13], we consider several variants of the core trail by changing some of the message constraints in the last steps (in particular, we flip the sign of some message bits), and we relax the last steps to reach a larger set of output difference. However, we do this more exhaustively than Stevens: he only describes a set $\mathcal{D}$ of size 192 with cost at most $1.15 \cdot C_{\text{block}}$, but we found a set of size 8768 with cost at most $8 \cdot C_{\text{block}}$, including 576 values with cost at most $1.15 \cdot C_{\text{block}}$. In particular, this directly leads to an improvement of the single-block chosen-prefix collision from [Ste13], with complexity roughly $\sqrt{\pi \cdot 2^{160}/8768} \approx 2^{74.3}$, rather than $\sqrt{\pi \cdot 2^{160}/192} \approx 2^{77}$.

Building the graph $\mathcal{G}$. We first build a graph with all nodes that are reachable with a path of cost at most 24 C_{block} and up to 10 blocks. This results in a graph with $2^{36.2}$ nodes, which requires 2 TB of storage (storing only the nodes and their cost). Using the bi-directional method above, our simulations show that there are around 2^{38} differences δ that can be used in this attack, with at most 11 blocks. Further simulations show that the average cost from a node in the graph to a collision is around 2 C_{block} when using implicit edges.

3.2.3 Running the attack in practice

Running an attack on this scale is a significant challenge. Following previous work [KPS15; SBK+17], we use GPUs for this computation, because they offer a better performance/price ratio.

Overall, for the attack parameters chosen, the birthday part costs about $2^{62.05}$ SHA-1 computations, while the near-collision part is expected to require 1 C_{block} for the last block, and 1 C_{block} in total for all the previous blocks. The total complexity is estimated as equivalent to $2^{63.5}$ SHA-1 evaluations on a GTX 1060 GPU, less than four times higher than the cost of an identical-prefix collision. Since SHA-1 is a 160-bit hash function, the expected security level for (CP) collision attacks is 2^{80}.

A GPU cluster. When running this computation in 2019, we originally estimated that our attack would cost around US$ 160k by renting GPUs from a cloud provider such as Amazon or Google (using spot or preemptible prices). However, since our computations do not require communication between the GPUs, nor fancy inter-GPU task scheduling, we instead rented cheaper GPUs from providers that use gaming or mining cards in consumer-grade PCs, rather that the datacenter-grade hardware used by big cloud providers. Services like gpuserversrental.com rented GTX 1060 or GTX 1080 GPUs for a price below 5 cents per month per CUDA core; this translates to a total cost around US$ 75k to compute a chosen-prefix collision.

Our cluster was made of 150 machines with 6 GPU each (with a mix of GTX 1060 3G, and GTX 1060 6G), and one master node with two 2 TB hard drives in a RAID configuration. The master node had a Core i7 CPU, but the GPU nodes had low-end Pentium or Celeron CPU with two cores.

Running the attack. The birthday phase is relatively straightforward, using the parallel collision search of van Oorschot and Wiener [vW99].

The near-collision phase is very technical and very complex. Every time a block is found, we have to prepare the search for the next block. This first requires traversing the graph $\mathcal{G}$ to find the parameters for the next block, and to generate a new non-linear trail for the early steps. We used tools similar to [DR06], which take a lot of parametrization and trial-and-error to have a proper non-linear trail that fits nicely with the core differential

path. This was automated to some extent, but still took between a few hours and a few days of manual work to prepare for each block.

After two months of computations, we successfully obtained a CP collision for SHA-1. Our collision uses only 9 near-collision blocks because some implicit edges correspond to shortcuts in the graph.

3.3 Impersonation attack against the PGP Web-of-trust

Our demonstration of a chosen-prefix collision targets the PGP/GnuPG Web of Trust. More precisely, our goal is to create two PGP keys with different UserIDs, so that key B is a legitimate key for Bob (to be signed by the Web of Trust), but the signature can be transferred to key A which is a forged key with Alice's ID. This will succeed if the hash values of the identity certificates collide, as in previous attacks against X.509 MD5-based certificates [SLW07; SSA+09]. Moreover, due to details of the PGP/GnuPG certificate structure, our attack can reuse a single CP collision to target arbitrary users Alice and Bob: for each victim, the attacker only needs to create a new key embedding the collision, and to collect a SHA-1 signature. This is arguably the first practical attack against a real-world security application using weaknesses of SHA-1.

3.3.1 Exploiting a chosen-prefix collision

We focus on the identity certificates that will be hashed and signed. Following [RFC4880], the hash computation done during certificate signing includes the public key packet, then a UserID or user attribute packet, and finally a signature metadata packet and a trailer. Our attack uses two public keys of different sizes, so that the remaining fields to be signed are misaligned, and we can hide the UserID of key A in another field of key B. Following RFC 4880, the signature metadata packet is protected by a length value at the beginning *and at the end*, so that we have to use the same signature metadata packet in key A and key B (we cannot stuff data in the hashed subpacket). Therefore, we can only play with the UserID and/or user attribute packets. Still, a user attribute packet with a JPEG image gives us enough freedom to build colliding certificates, because typical JPEG readers ignore any bytes after the End of Image marker (`ff d9`). This gives us some freedom to stuff arbitrary data in the certificate.

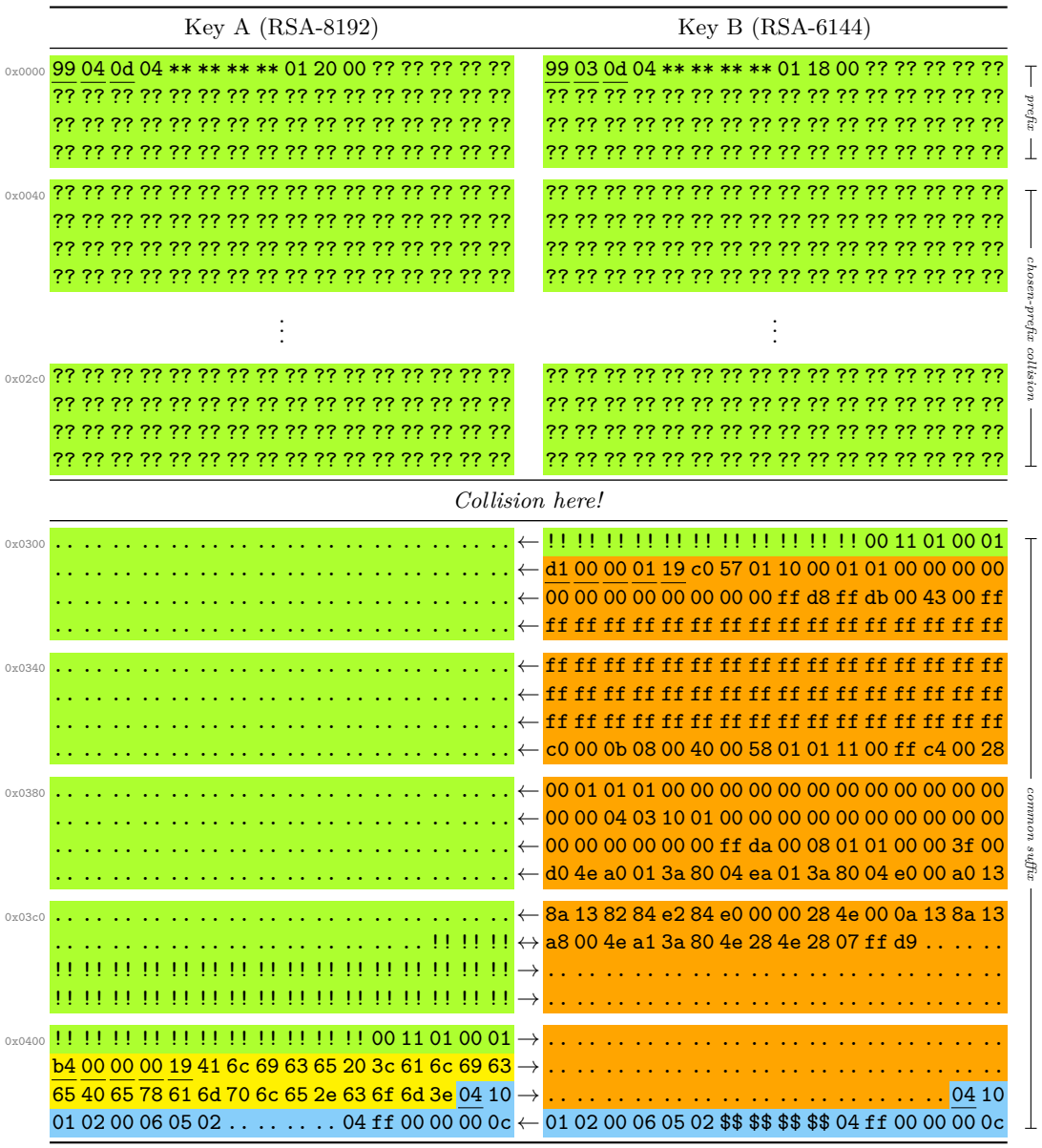
More precisely, we build keys A and B as follows. Key A contains an 8192-bit RSA public key, and a UserID field corresponding to Alice. On the other hand, key B contains a 6144-bit RSA public key, and a user attribute containing a JPEG image.

Content of identity certificates. Figure 3.9 shows a template of the values included in the identity certificate: those values are hashed when signing a key, and we want the two hashes to collide. In this example, the UserID field of key A contains "`Alice <alice@example.com>`", and the image in key B is a valid JPEG image that will be padded with junk data after the End of Image marker. The real JPEG file is 181 bytes long[1] (from `ff d8` to `ff d9`), and it is padded with 81 bytes, so that the file included in the key is 262 bytes long (here the padding includes 46 bytes corresponding to the end of the modulus of key A, 5 bytes corresponding to the exponent of key A, and 30 bytes corresponding to Alice's UserID).

After computing the CP collision, we obtain a pair of keys with colliding certificates:

- https://sha-mbles.github.io/alice.asc
- https://sha-mbles.github.io/bob.asc

[1] Building a JPEG image smaller than 256 bytes is not easy, but it is possible.



```
               Key A (RSA-8192)                                      Key B (RSA-6144)
0x0000  99 04 0d 04 ** ** ** ** 01 20 00 ?? ?? ?? ?? ??     99 03 0d 04 ** ** ** ** 01 18 00 ?? ?? ?? ?? ??
        ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??     ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
        ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??     ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
        ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??     ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??

0x0040  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??     ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
        ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??     ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
        ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??     ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
        ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??     ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
                              ⋮                                                   ⋮
0x02c0  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??     ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
        ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??     ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
        ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??     ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
        ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??     ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
                                           Collision here!
0x0300  .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..  ←  !! !! !! !! !! !! !! !! !! !! !! 00 11 01 00 01
        .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..  ←  d1 00 00 01 19 c0 57 01 10 00 01 01 00 00 00 00
        .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..  ←  00 00 00 00 00 00 00 00 ff d8 ff db 00 43 00 ff
        .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..  ←  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff

0x0340  .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..  ←  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
        .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..  ←  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
        .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..  ←  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
        .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..  ←  c0 00 0b 08 00 40 00 58 01 01 11 00 ff c4 00 28

0x0380  .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..  ←  00 01 01 01 00 00 00 00 00 00 00 00 00 00 00 00
        .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..  ←  00 00 04 03 10 01 00 00 00 00 00 00 00 00 00 00
        .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..  ←  00 00 00 00 00 00 ff da 00 08 01 01 00 00 3f 00
        .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..  ←  d0 4e a0 01 3a 80 04 ea 01 3a 80 04 e0 00 a0 13

0x03c0  .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..  ←  8a 13 82 84 e2 84 e0 00 00 28 4e 00 0a 13 8a 13
        .. .. .. .. .. .. .. .. .. .. .. .. .. !! !! !!  ↔  a8 00 4e a1 3a 80 4e 28 4e 28 07 ff d9 .. .. ..
        !! !! !! !! !! !! !! !! !! !! !! !! !! !! !! !!  →  .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
        !! !! !! !! !! !! !! !! !! !! !! !! !! !! !! !!  →  .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..

0x0400  !! !! !! !! !! !! !! !! !! !! !! 00 11 01 00 01  →  .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
        b4 00 00 00 19 41 6c 69 63 65 20 3c 61 6c 69 63  →  .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
        65 40 65 78 61 6d 70 6c 65 2e 63 6f 6d 3e 04 10  →  .. .. .. .. .. .. .. .. .. .. .. .. .. .. 04 10
        01 02 00 06 05 02 .. .. .. .. 04 ff 00 00 00 0c  ←  01 02 00 06 05 02 $$ $$ $$ $$ 04 ff 00 00 00 0c
```

Figure 3.9: Construction of colliding OpenPGP identity certificates.
The colors correspond to the packets hashed when computing the signature: first, the public key packet (with header), then the UserID or user attribute, and finally the signature packet and trailer. Arrows show when a value is chosen in one key and copied to the other.
We use the following symbols:
`01` Value fixed by the specifications, or chosen in advance (length of fields, UserID, ...)
`??` Bytes determined by the chosen-prefix collision algorithm
`!!` Bytes selected after finding the collision, to generate an RSA modulus
`..` Bytes that are copied from the other certificate
`**` Time-stamps chosen by the attacker
`$$` Time-stamp chosen by the signer
Underlined values correspond to packet headers (type and length).

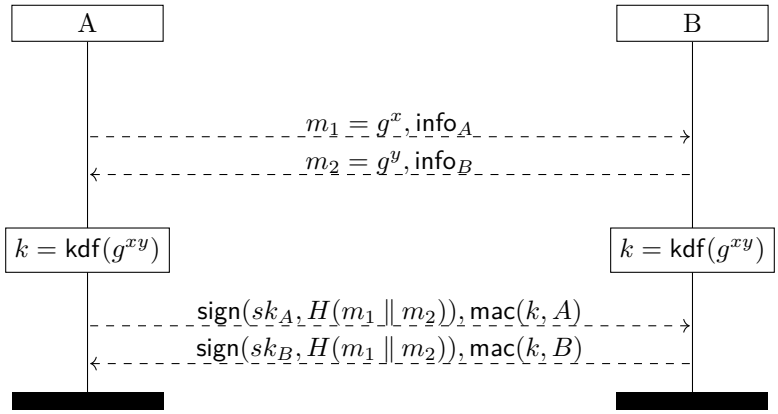


Figure 3.10: SIGMA': A mutually-authenticated key exchange protocol.

3.4 Transcript collision attacks

Many popular cryptographic protocols have been defined using hash functions MD5 and SHA-1, which are vulnerable to collision attacks and chosen-prefix collision attacks.

Practitioners commonly believe that collisions only affect non-repudiable signatures (like certificates), but that signatures and MACs used within protocols are safe as long as they include unpredictable contents, such as nonces [RFC4894; RFC4270]. In these cases, protocol folklore says that a second preimage attack would be required to break these protocols, and such attacks are still considered hard, even for MD5.

Conversely, theoretical cryptographers routinely assume collision-resistance in proofs of security for these protocols. For example, various proofs of TLS [JKS+12; KPW13; GKS13] assume collision-resistance even though the most popular hash functions used in TLS at the time were MD5 and SHA-1. Whom shall we believe? Either it is the case that cryptographic proofs of these protocols are based on too-strong (i.e. false) assumptions that should be weakened, or that practitioners are wrong and collision resistance is required for protocol security.

In this section, we clarify this situation by investigating the use of hash functions in the authenticated key exchanges. We describe a generic class of attacks called *transcript collision* attacks, demonstrating that, contrary to common belief, collisions *can* be used to break fundamental security guarantees of these protocols.

3.4.1 Authenticated Key Exchange

Authenticated Key Exchange (AKE) protocols are executed between two parties, usually called client and server or initiator and responder, in order to establish a shared session key that can be used to encrypt subsequent messages. A typical example is the SIGMA' protocol depicted in Figure 3.10. This protocol is a variant of the basic SIGMA (sign-and-mac) protocol from [Kra03] which served as the inspiration for the key exchanges used in many protocols including IKE, OTR, and JFK.

In SIGMA', the initiator A first sends a message m_1 to B, consisting of a Diffie-Hellman public value g^x, along with some protocol-specific parameters info_A that may include, for example, a nonce, a protocol version, a proposed ciphersuite, etc. B responds with a message m_2 containing its own Diffie-Hellman public value g^y and some parameters info_B. A and B have now completed an anonymous Diffie-Hellman exchange and can compute the shared secret g^{xy} and use it to derive the session key. However, before using the session key, they authenticate each other by exchanging digital signatures over the protocol transcript $H(m_1 \| m_2)$ using their long-term signing keys (sk_A, sk_B). (Digital signature

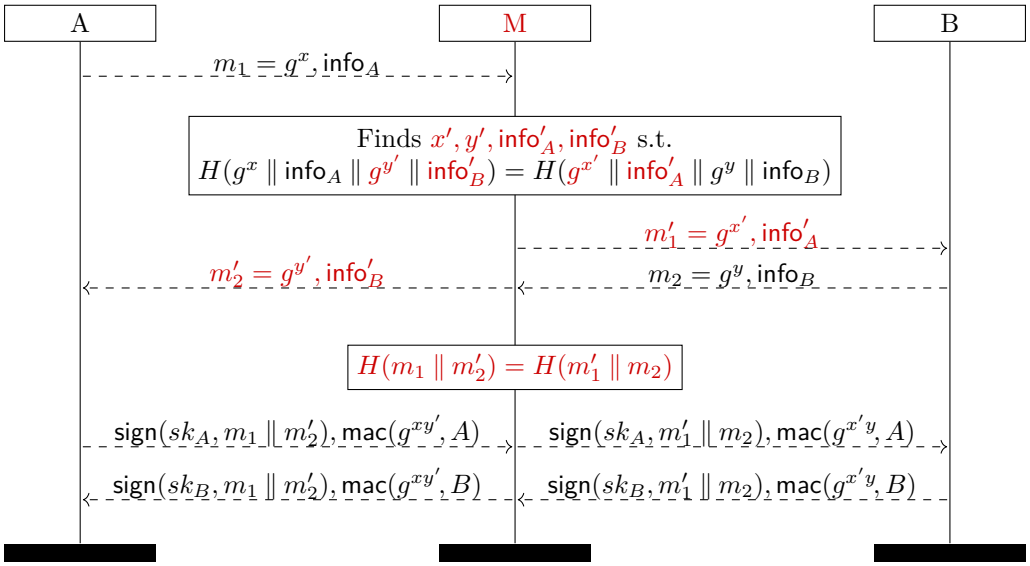


Figure 3.11: Man-in-the-middle credential forwarding attack on SIGMA'. The attacker creates a *transcript collision* by tampering with the messages shown in red. At the end of the protocol, the client and server have seemingly authenticated each other, but the attacker knows both connection keys, and hence can read or write any data.

algorithms typically hash their arguments before signing them, and we have chosen to make this hashing explicit in our presentation of SIGMA'.) By signing the transcript, A and B verify that they agree upon all the elements of the key exchange, and in particular, that a network attacker has not tampered with the messages. Finally, A and B also prove to each other that they know the session key g^{xy} by exchanging MACs computed with this key over their own identities.

Like other AKE protocols, SIGMA' aims to prevent message tampering, peer impersonation, and session key leakage, even if the network and other clients and servers are under the control of the adversary. Formally, authenticating the transcript guarantees *matching conversations*, that is, that the two parties agree on each other's identity and other important protocol parameters.

3.4.2 Transcript collision attacks

The alert reader will notice that SIGMA' *does not* in fact guarantee that A and B agree on the message sequence $m_1 \parallel m_2$; it only guarantees that they agree on the *hash* of this sequence. What if a network attacker were to tamper with the messages, so that A and B see different message sequences but the hashes of the two sequences is the same? In that case, the protocol will proceed to completion but the integrity and authentication guarantees no longer hold.

Figure 3.11 illustrates such an attack in the man-in-the-middle setting. The attacker intercepts messages sent between A and B. She sends her own message $m'_1 = g^{x'} \parallel \mathsf{info}'_A$ to B and her own response $m'_2 = g^{y'} \parallel \mathsf{info}'_B$ to A. Suppose she can choose these messages such that the authenticated transcripts match:

$$H(m_1 \parallel m'_2) = H(m'_1 \parallel m_2)$$

We call this a *transcript collision*. Now, the attacker can simply forward A's signature over this transcript to B and vice versa. A and B will accept the signatures since the hashed transcripts match and the signing keys are correct. However, the attacker knows the session

keys $(g^{x'y}, g^{xy'})$ on both connections (since she knows x', y'). Hence, the attacker has fully hijacked both connections and can now send messages to B pretending to be A and to A pretending to be B. This is an *impersonation* attack that breaks peer authentication.

If the boundaries between the messages m_1 and m_2 are not clearly demarcated, there are a number of trivial attacks that can ensure that $m_1 \parallel m_2' = m_1' \parallel m_2$ with no need for hash collisions. In the following examples, we will assume that each message (and each message field) is prefixed with its length.

3.4.3 A generic transcript collision

The main challenge in implementing the attack in Figure 3.11 is that the attacker has to compute the messages m_1' and m_2' *after* receiving m_1 but *before* the responder has sent her response m_2. The feasibility of the attack depends on the contents and formats of these messages.

Suppose the responder B always sends the same message m_2 for every request; that is, she uses the same (static) Diffie-Hellman value g^y and same parameters info_B. This situation occurs, for example in protocols like QUIC, where the server uses a static configuration. In that case, after receiving m_1, the attacker can compute a transcript collision by finding $x', y', \mathsf{info}_A', \mathsf{info}_B'$ such that $H(m_1 \parallel m_2') = H(m_1' \parallel m_2)$. This requires $2^{n/2}$ hash evaluations, because of the birthday paradox.

3.4.4 A chosen-prefix transcript collision

We now consider a more efficient attack that works even when B sends an unpredictable m_2 containing a fresh (ephemeral) Diffie-Hellman value g^y and a previously unknown info_B. However, we assume that the length of m_2 (ℓ_m) is fixed and known to the attacker. Moreover, suppose that in the second message of SIGMA', info_B is allowed to have arbitrary length and arbitrary contents. That is, even if info_B has junk data at the end, A will accept the message. Specifically, suppose that $\mathsf{info}_B = \mathsf{len}_B \parallel \mathsf{data}_B$ where data_B is opaque data that will be ignored by A. Such "collision-friendly" messages exist in TLS, IKE, and SSH.

Under this condition, the attacker can compute a transcript collision by constructing a chosen-prefix collision (C_1, C_2) of ℓ_1 and ℓ_2 bytes respectively, such that:

$$H(m_1 \parallel \underbrace{[g^{y'} \parallel \overbrace{\mathsf{len}_B' \parallel C_1 \parallel -}^{\mathsf{info}_B'}]}_{m_2'}) = H(\underbrace{[g^{x'} \parallel \overbrace{C_2}^{\mathsf{info}_A'}]}_{m_1'})$$

where $\mathsf{len}_B' = \ell_1 + \ell_m$. Note that we have left empty space (written $-$) of size ℓ_m bytes that still needs to be filled after C_1 in info_B'.

After receiving m_1 from A and computing C_1, C_2, the attacker sends m_1' to B. When B responds with m_2 (of size ℓ_m bytes), she stuffs m_2 at the end of info_B' (in place of $-$) and sends m_2' to A. Due to the collision extension property, we have:

$$H(m_1 \parallel \underbrace{[g^{y'} \parallel \overbrace{\mathsf{len}_B' \parallel C_1 \parallel m_2}^{\mathsf{info}_B'}]}_{m_2'}) = H(\underbrace{[g^{x'} \parallel \overbrace{C_2}^{\mathsf{info}_A'}]}_{m_1'} \parallel m_2)$$

That is, the attacker has obtained a transcript collision and the impersonation attack succeeds.

Table 3.1: Complexity of collision attacks (SHA-1 equivalents on GTX 1060 GPU)

Function	MD5 ($n = 128$)	SHA-1 ($n = 160$)		MD5 ‖ SHA-1
		previous	§ 3.2	(w/ multicollision)
Collision	2^{16} [SSA+09]	$2^{64.7}$ [SBK+17]	$2^{61.6}$	$2^{67.6}$
CP collision	$2^{39.1}$ [SSA+09]	$2^{77.1}$ [Ste13]	$2^{63.5}$	$2^{67.6}$

3.4.5 Application to TLS, IKE and SSH

The transcript-collision attack exploits hash collisions in combination with flexible protocol-specific message formats. This trick can be applied to several mainstream protocols such as TLS, IKE, and SSH.

In [BL16b] (Appendix E), we demonstrate how a man-in-the-middle attacker can impersonate TLS 1.2 clients, TLS 1.3 servers, and IKEv2 initiators. We also show how a network attacker can downgrade TLS 1.1 and SSH-2 connections to use weak ciphers. We have implemented proofs-of-concept exploit demos for three of these attacks to demonstrate their practicality, and provide attack complexities for the others. We believe that ours are the first hash collision-based attacks on the cryptographic constructions within these protocols. This settles the debate on whether the security of mainstream cryptographic protocols depends on collision resistance. We recommend that weak hash functions like MD5 and SHA-1 should be disabled from Internet protocols.

3.5 Conclusion

Chosen-prefix collisions have first been demonstrated on MD5 in 2007 [SLW07]. In this chapter we demonstrated them on SHA-1 for the first time. We summarize the complexity of known attacks in Table 3.1. We also provide several new applications of CP collisions, clearly showing that MD5 and SHA-1 signatures can be abused in practice. The transcript-collision attacks are hard to deploy in the wild because the CP collision must be computed online; on the other hand, the PGP impersonation attack is easy to mount because the CP collision is a precomputation, independent of the target identity.

Responsible disclosure. Our transcript-collision attacks on TLS were responsibly disclosed under the name SLOTH[2] in 2015 (`CVE-2015-7575`) and have resulted in security updates to several TLS libraries. Our chosen-prefix attack on SHA-1 was also responsibly disclosed under the name Shambles[3] in 2019 (`CVE-2019-14855`). In particular, we contacted the authors of GnuPG and OpenSSL.

3.5.1 SHA-1 usage and deprecation

SHA-1 has been known to be broken since 2005 [WYY05]. NIST has been relatively fast to deprecate SHA-1 usage for signatures:

> “ *Federal agencies should stop using SHA-1 for digital signatures, digital time stamping and other applications that require collision resistance as soon as practical, and must use the SHA-2 family of hash functions for these applications after 2010.* ”
>
> — NIST, March 15, 2006

[2] https://www.mitls.org/pages/attacks/SLOTH
[3] https://sha-mbles.github.io

However, SHA-1 was still used in many different contexts when we demonstrated our chosen-prefix collision attack in early 2020, including within signatures. In late 2020, NIST announced than it would retire SHA-1 for all usages in 2030, due in part to our chosen-prefix collision attack.[4]

X.509 certificates. The CA/Browser Forum decided to sunset SHA-1 in October 2014, and its members were not supposed to issue SHA-1 certificates after 2016. Web browsers have enforced similar rules, and major browsers started to reject SHA-1 certificates in early 2017. Figure 1.4 shows hash function usage in the certificates of top websites: SHA-1 was still used until 2016 and stopped being used in early 2017, around the time the first collision was announced [SBK+17].

For non-web clients, SHA-1 certificates have been supported longer in some applications. For instance, OpenSSL has deprecated SHA-1 signatures with version 3.0.0 in September 2021.

PGP / GnuPG. There are currently two supported branches of GnuPG: GnuPGv1 is the "legacy" (or "classic") branch, and GnuPGv2 is the "modern" branch. The first version of GnuPGv2 dates back to 2006, and the "legacy" branch is no longer recommended, but the transition took a long time. In particular, GnuPGv1 was still the default version in Fedora 29 (released in October 2018), and in Ubuntu 16.04 LTS (supported until April 2021).

GnuPG supports many different algorithms, including SHA-1. Moreover, SHA-1 is the default algorithm for identity certification in GnuPGv1. This is why we targeted PGP in our demonstration of chosen-prefix collisions. After we disclosed our results to the GnuPG team, SHA-1 signatures are no longer trusted in the GnuPGv2 branch.

The current draft of the next PGP standard [WHW+23] deprecates MD5 and SHA-1, and defines a new signature format adding a salt at the beginning of the hashed data. This prevents attacks were one party generates a collision and asks another party to sign it.

TLS handshake. In TLS 1.0 and 1.1, the handshake is hashed with the concatenation of SHA-1 and MD5. Using the multicollision attack from Joux [Jou04], computing a CP collision for MD5 ‖ SHA-1 is not much harder than for SHA-1. This makes those versions vulnerable to transcript-collision attacks. Figure 1.5 shows TLS usage for top websites: TLS 1.0 and 1.1 are being phased out, but they were still used when we demonstrated our attack. TLS 1.0 and 1.1 have been deprecated in 2021 [RFC8996], mostly because of SHA-1 weaknesses.

In TLS 1.2, the hash function used is configurable. In 2015, many servers and client supported MD5, and some used it in normal usage: this was the default configuration of Java clients and servers. MD5 has been disabled in many libraries after disclosing our attack. In late 2019, the vast majority of clients and servers supported SHA-1, and some servers actually *preferred* to use SHA-1, even when the client offers better algorithms.[5,6] Usage of MD5 and SHA-1 has been deprecated in 2021 [RFC9155].

Modern clients (such as web browsers and OpenSSL) have removed support for TLS 1.0 and 1.1, as well as SHA-1 support to sign the handshake.

DNSSEC. SHA-1 was still used in DNSSEC, with 18% of the top-level domains using SHA-1 in early 2020.[7] Since DNSSEC signatures include user-supplied content, CP collisions could potentially be used to attack the DNSSEC system.

[4] https://csrc.nist.gov/Projects/Hash-Functions/NIST-Policy-on-Hash-Functions

[5] http://web.archive.org/web/20191227174651/https://censys.io/domain/report?field=443.https.tls.signature.hash_algorithm

[6] http://web.archive.org/web/20191227174551/https://censys.io/domain?q=443.https.tls.signature.hash_algorithm%3Asha1

[7] https://www.dns.cam.ac.uk/news/2020-02-14-sha-mbles.html

4. Generic Attacks Against Hash-Based MACs

This chapter is based on articles [LPW13] and [DL14; DL16], found in Appendix F and G.

Message Authentication Codes (MACs) are an important type of symmetric cryptography primitives, used to verify the integrity and authenticity of messages. The sender of a message uses a MAC to compute a tag from the message and a shared secret key. The tag is appended to the message and the receiver can recompute the tag using the key, and reject the message when it does not match the received tag. Common MAC algorithms are built from block ciphers (*e.g.* `CBC-MAC`, `PMAC`), from hash functions (*e.g.* `HMAC`, `NMAC`), or from universal hash functions (*e.g.* `GMAC`, `Poly1305`, `UMAC`).

4.1 Security of MAC algorithms

The main security requirement of a MAC is the resistance to forgery: it should be hard for an adversary to forge a valid tag without knowing the key. We define two variants of forgery attacks, assuming oracle access to the MAC with the secret key:

Existential forgery. After querying the oracle to obtain the tags of some chosen messages, the adversary must predict the tag of a message that has not been queried.

Universal forgery. The adversary is given a challenge message C before interacting with the oracle. She can query any message excepted C, and must predict the tag of C.

It should also be impossible to recover the secret key excepted by exhaustive search. In addition, cryptanalysts have studied state-recovery attacks, where the adversary attempts to recover the internal state of the MAC.

4.1.1 Generic forgery attack against iterative MACs

Preneel and van Oorschot described a generic attack against all deterministic iterated MACs with a state size of n bits [Pv95] based on collisions. After $2^{n/2}$ queries with small messages of the same length, we expect that two messages reach the same internal state. If the message pair (M, M') produces an internal collision, then for any suffix S the pair $(M \parallel S, M' \parallel S)$ also collides in the tag: $\texttt{MAC}(M \parallel S) = \texttt{MAC}(M' \parallel S)$. Therefore, we can create a forgery by asking for $\texttt{MAC}(M \parallel S)$ and using it to predict $\texttt{MAC}(M' \parallel S)$.

This attack is similar to the collision extension property of hash functions, but the attacker does not have access to the internal state, because a MAC is a keyed function. However, the internal collision can be detected because it implies a collision in the tag. For some MACs there are also false-positive pairs colliding on the tag but not on the internal state, but they are filtered out by testing the forgery attack.

4.1.2 Security loss beyond the birthday bound

Even though all deterministic iterated MACs have essentially the same security bound, the impact of the attacks can be quite different. For instance, the Envelope MAC [Tsu92] and the Sandwich MAC [Yas07] are two similar MAC constructions based on hash functions with an n-bit internal state:

$$\texttt{Envelope}_k(M) = H(k \parallel M \parallel k)$$
$$\texttt{Sandwich}_k(M) = H(\mathsf{pad}(k) \parallel \mathsf{pad}(M) \parallel \mathsf{pad}(k))$$

where pad pads its input to an integer number of blocks. Both modes are secure up to the birthday bound, and the generic forgery attack matches the bound. However, there is a more complex key-recovery attack against the envelope MAC with birthday complexity [Pv96].

Key-recovery attack against the Envelope MAC. For simplicity, we assume that the hash function uses n-bit message blocks, and that the size of the key is also n bits. The attacker considers n-bit messages (M, M'), with the last bit equal to zero ($M[n-1] = M'[n-1] = 0$). After querying $2^{n/2}$ messages she expects a collision in the internal state:

$$h^*(k \parallel M) = h^*(k \parallel M')$$

She detects the hash collision as a MAC collision ($\texttt{Envelope}_k(M) = \texttt{Envelope}_k(M')$), and she verifies that it is an internal state collision with a few extra queries.

Next, she queries the MAC of $n-1$-bit messages $M[0..n-2]$ and $M'[0..n-2]$. If $k[0] = 0$, then the tags collide because the first two blocks processed by the hash function are the same as in the internal collision $h^*(k \parallel M) = h^*(k \parallel M')$:

$$k \parallel M\,[0..n-2] \parallel k[0] = k \parallel M\,[0..n-2] \parallel 0 = k \parallel M$$
$$k \parallel M'[0..n-2] \parallel k[0] = k \parallel M'[0..n-2] \parallel 0 = k \parallel M'$$

The collision is preserved by the final block, because its content is the same for both messages ($k[1..n-1]$).

On the other hand, if $k[0] = 1$, the tags are different with high probability because we have two different values for the second block ($M[0..n-2] \parallel 1$ and $M'[0..n-2] \parallel 1$) that are not known to collide. Using this technique, the adversary recovers the key iteratively (bit-by-bit or byte-by-byte), building a special collision corresponding to each successive partial guess. She obtains a key-recovery attack with complexity $\tilde{\mathcal{O}}(2^{n/2})$.

This attack does not apply to the Sandwich MAC, because the padding after the message prevents mixing message bits and key bits in the same block. This shows that modes with similar security proofs can have different security degradation beyond the birthday bound.

4.2 Hash-based MACs

In this chapter we study MAC algorithms based on a hash function such as HMAC [BCK96], one of the most widely used MACs. Using a hash function H, HMAC is defined as

$$\texttt{HMAC}(k, M) = H(k \oplus \texttt{opad} \parallel H(k \oplus \texttt{ipad} \parallel M)),$$

with ipad and opad two fixed constants of size one block. More generally, we consider a class of iterative designs based on a family of compression functions $h^{(i)}$ and a finalization function g, following Chapter 2. For simplicity, we assume that the internal state and

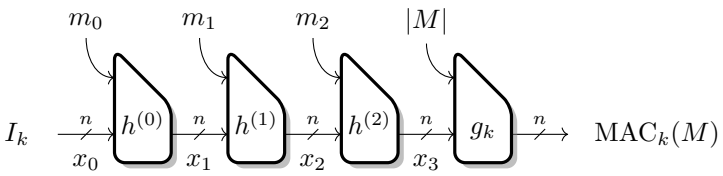


Figure 4.1: Hash-based MAC. Only the initial value and the final transformation are keyed.

the tag are both n-bit long. The message M is divided into L blocks x_i, and the tag τ is computed as (see Figure 4.1):

$$x_0 = I_k \qquad x_{i+1} = h^{(i)}(x_i, m_i) \qquad \tau = g_k(x_L, |M|).$$

The message processing updates the internal state starting from a key-dependent value I_k, and the output is produced with a key-dependent finalization function g_k. On the other hand the state update does not depend on the key.

Our description covers `HMAC` [BCK96], `Sandwich-MAC` [Yas07] and `Envelope-MAC` [Tsu92] with common narrow-pipe hash functions. In particular, we consider that the finalization function depends on the message length $|M|$, because many hash functions use MD strengthening. When the hash function uses the Merkle-Damgård mode, the compression function is the same at every step, and we denote it as h ($\forall i, h^{(i)} = h$).

We know that iterated hash functions are indifferentiable up to the birthday bound, assuming that the compression function is ideal [CDM+05]. This also implies that hash-based MACs are secure up to the birthday bound.

Our results. In the following we study the security of hash-based MACs above the birthday bound. We give generic state-recovery attacks with complexity $\tilde{\mathcal{O}}(2^{n/2})$ when the hash function follows the Merkle-Damgård mode (Section 4.3) and $\tilde{\mathcal{O}}(2^{4n/5})$ for the more general HAIFA mode (Section 4.4). This contradicts previous beliefs that state-recovery attacks should have complexity 2^n. We also extend these to key-recovery attacks when the hash function uses an internal checksum (Section 4.5).

4.3 State-recovery attack using cycles

Our first attack uses long messages repeating a fixed block β (for instance, $\beta = 0$). We assume that the hash function uses the Merkle-Damgård construction, so that the compression function h is the same at every step. When evaluating the MAC on such a message, we actually iterate a fixed function $\phi : x \mapsto h(x, \beta)$; our goal is to exploit properties of the functional graph of ϕ. We travel in the graph by querying the oracle using consecutive β message blocks. However, since the key is unknown, we do not have access to the internal state; we can only detect collisions because they are preserved by the finalization function.

4.3.1 Forgery

We start with a forgery attack based on collisions. This attack exploits the cycle length of the main component of ϕ. Since ϕ is a public function, we identify the main component and its main cycle with $\mathcal{O}(2^n/2)$ operations; we denote its length as ℓ.

Assuming that ϕ behaves as a random function, the expected tail length (the distance to the cycle) is $\sqrt{\pi 2^{n-3}}$ and the expected cycle length is also $\sqrt{\pi 2^{n-3}}$ [FO90]. Moreover, the main component has expected size 0.76×2^n [FS09, §VII.14].

Naive attack. We first assume that no length padding is used in the hash function, so that the finalization function g does not depend on the message length. In this case, it is quite easy to obtain a collision, and therefore a forgery. We consider the following messages

$$M_1 = \beta^{2^{n/2}} \qquad M_2 = \beta^{2^{n/2}+\ell}$$

With constant probability, the initial value I_k is in the main component, and the sequence of $2^{n/2}$ β blocks is sufficiently long to enter the cycle. In this case, the two MAC outputs will collide.

Dealing with MD strengthening. In order to deal with the MD strengthening, we must generate two messages of the same length. We use the following trick: we start from a message $M = \beta^{2^{n/2}} \parallel \alpha \parallel \beta^{2^{n/2}}$ with an arbitrary block $\alpha \neq \beta$. With constant probability, the internal state reaches the main cycle after the first series of β, and again after the second series of β. In this case, we can add ℓ β blocks in the first or the second series to obtain two colliding messages of the same length (see Figure 4.2):

$$M_1 = \beta^{2^{n/2}} \parallel \alpha \parallel \beta^{2^{n/2}+\ell} \qquad M_2 = \beta^{2^{n/2}+\ell} \parallel \alpha \parallel \beta^{2^{n/2}}$$

Complexity analysis. The attack has complexity $\mathcal{O}(2^{n/2})$ and succeeds if

1. I_k is in the main component of ϕ;
2. The cycle is reached after less than $2^{n/2}$ blocks β;
3. The state reached after α is also in the main component of ϕ;
4. The cycle is reached after less than $2^{n/2}$ blocks β.

Events 1 and 3 have probability about 0.76, while events 2 and 4 have probability at least $1/2$ given that the expected tail length is shorter than $2^{n/2}$. Overall the success probability is at least 0.14. The attack can be repeated with a random prefix block if needed.

4.3.2 State recovery

We now extend the forgery attack into a state-recovery attack. We use an additional property of the graph of a random mapping: the graph contains a giant tree of expected size 0.48×2^n [FS09, §VII.14]. Therefore, starting from a random node in the graph, there is a constant probability to enter the cycle at one specific node: the root of the giant tree. Since ϕ is a public function, we identify this node offline with complexity $\mathcal{O}(2^{n/2})$.

When the forgery attack above succeeds, there is a constant probability that the first cyclic node is exactly the root of the giant tree. Therefore, we obtain a state-recovery attack by searching for the smallest length z such that the cycle is reached after processing β^z. If the cycle is reached, we know that the following two messages have a constant probability to have the same MAC:

$$M_1 = \beta^z \parallel \alpha \parallel \beta^{2^{n/2}+\ell} \qquad M_2 = \beta^{z+\ell} \parallel \alpha \parallel \beta^{2^{n/2}}$$

In order to avoid false negatives, we repeat queries with several choices of α for each z, to obtain a high-selectivity test. Using a binary search, we need only $\mathcal{O}(n)$ queries to find the smallest z reaching the cycle. With constant probability, the state after processing the message β^z is the root of the giant tree.

Complexity analysis. The attack has complexity $\tilde{\mathcal{O}}(2^{n/2})$ and succeeds with constant probability.

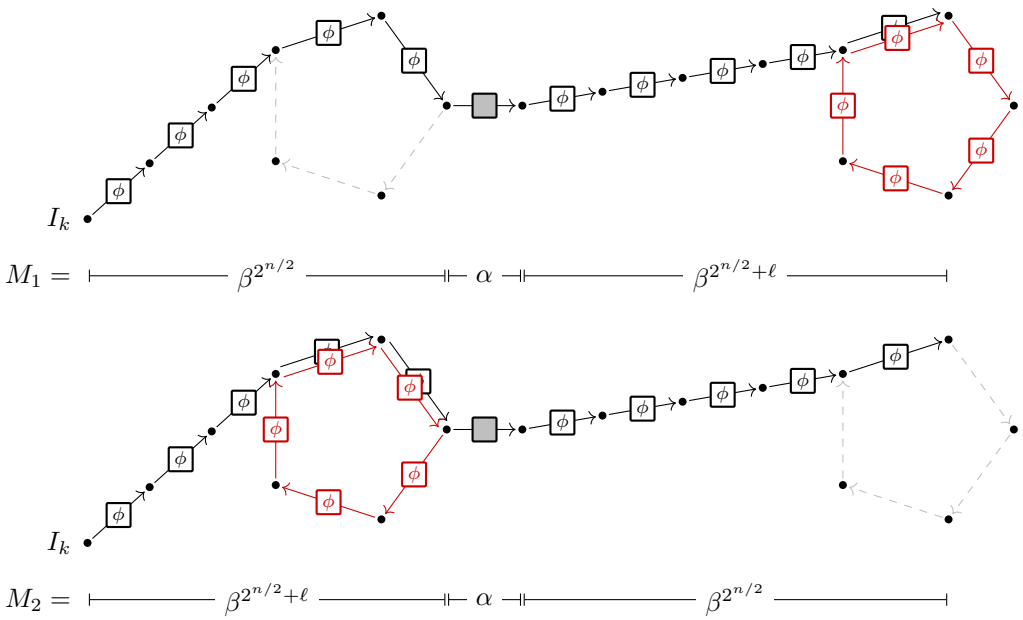


Figure 4.2: Cycle-based collision.

4.4 State-recovery attack based on chains

We now propose an attack applicable to HAIFA hash functions, *i.e.* with different compression functions $h^{(i)}$ at each step, but we sometimes omit the step index for simplicity. Again, we use a fixed long message, but since we have different functions we cannot use properties of the functional graph. Instead, we study the entropy loss.

4.4.1 Entropy loss

When we iterate a fixed sequence of random functions, the image space is reduced at each step. More precisely, we have the following lemma:

Lemma 1. *Let $f_1, f_2, \ldots, f_{2^s}$ be a sequence of random functions over the set of 2^n elements, with $s \leq n/2$ and $g_i \triangleq f_i \circ \ldots \circ f_2 \circ f_1$. Then, the images of two arbitrary inputs to g_{2^s} collide with probability of about 2^{s-n},* i.e. $\Pr_{x,y}\left[g_{2^s}(x) = g_{2^s}(y)\right] = \Theta(2^{s-n})$.

In order to exploit this property, we fix a message C of length 2^s, and query the oracle with messages of the form $r \parallel C$. If we denote the function $x \mapsto h^{(i)}(x, C[i-1])$ as f_i $(1 \leq i \leq 2^s)$, the final state after processing $r \| C$ is in the image space of $g_{2^s} \triangleq f_{2^s} \circ \ldots \circ f_2 \circ f_1$. This set has properties similar to the root of the giant tree used in the previous attack: we can sample the set offline, it has reduced entropy, and we can build messages that reach this set online. In order to obtain a state-recovery attack, we need to match the unknown states obtained though online queries with the known states computed offline.

4.4.2 Using collision filters

A collision filter for an offline state x is a pair of blocks $b \neq b'$ such that $h(x, b) = h(x, b')$. Given a message M, we test whether the state obtained after processing M is equal to x by comparing the tags of $M \parallel b$ and $M \parallel b'$. Building the filter has complexity $\mathcal{O}(2^{n/2})$ and using it has complexity $\mathcal{O}(|M|)$.

Alternatively, we can build the filter online, and evaluate it on offline states. An online collision filter for a message M is a pair of blocks $b \neq b'$, such that $\texttt{MAC}(M \| b) = \texttt{MAC}(M \| b')$.

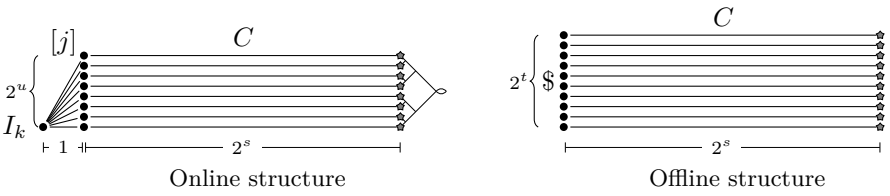


Figure 4.3: Chain-based state recovery. We detect a match between the gray points (⋆) using the diamond test built online.

Given a state x, we test whether the state obtained after processing M is equal to x by comparing $h(x, b)$ and $h(x, b')$. Building the filter has complexity $\mathcal{O}(|M| \times 2^{n/2})$ and using it has complexity $\mathcal{O}(1)$.

State recovery. We obtain a first attack using online collision filters:

1. (online) Fix a message C of length 2^s. Query the oracle with 2^u messages $M_j = [j] \| C$. Build a collision filter for each message M_j.
2. (offline) Starting from 2^t arbitrary starting points, iterate the compression function with the fixed message C.
3. (offline) Test each image point x obtained in Step 2 against each of the messages M_j. If a match is found, the state reached after M_j is x.

Complexity analysis. In Step 3, we match a set of size 2^u and a set of size 2^t. According to Lemma 1, each pair collides with probability 2^{s-n}. Therefore, the attack is successful with high probability if $t + u \geq n - s$.

The optimal trade-off is obtained with $u = s$ and $t = n - 2s$, with the following complexity:

Step 1: $2^{s+u+n/2} = 2^{n/2+2s}$ **Step 2:** $2^{s+t} = 2^{n-s}$ **Step 3:** $2^{t+u} = 2^{n-s}$

The optimal complexity is $\mathcal{O}(2^{5n/6})$ when $s = n/6$. More generally, we obtain an attack with complexity $\mathcal{O}(2^{n-s})$ when $s \leq n/6$.

4.4.3 Using a diamond filter

Building the filters is the bottleneck of the attack. In order to build filters for 2^u messages M_j more efficiently, we use the diamond structure of Kelsey and Kohno [KK06]. Since the construction of the diamond structure is based on collisions, we can build a diamond for the unknown states corresponding to online queries. This requires $\tilde{\mathcal{O}}(2^{(n+u)/2})$ queries, with messages of length $\mathcal{O}(|M_j|)$.

We obtain messages m_j such that all messages $M_j \| m_j$ reach the same state. Then we build a collision filter (b, b') for the final state of the diamond, and we obtain filters for all messages M_j. We test whether the state obtained after processing M_j is equal to x by comparing $h^*(x, m_j \| b)$ and $h^*(x, m_j \| b')$. Building the filter for 2^u messages has complexity $\mathcal{O}(|M_j| \times 2^{(n+u)/2})$, and using a single filter has complexity $\mathcal{O}(u)$.

State recovery. We obtain an improved attack using the online diamond filter (Figure 4.3):

1. (online) Fix a message C of length 2^s. Query the oracle with 2^u messages $M_j = [j] \| C$. Build a diamond filter for all messages M_j.

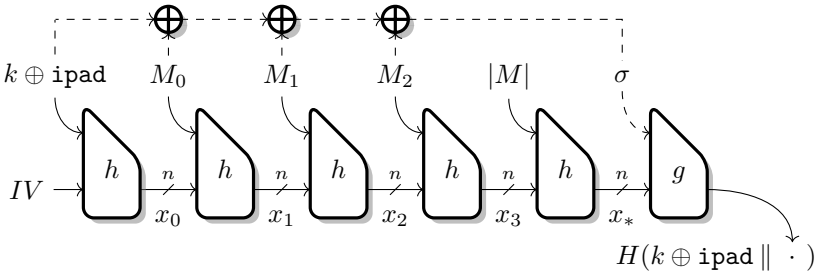


Figure 4.4: `HMAC` based on a hash function with a checksum (dashed lines) and a length-padding block. We only detail the first hash function call.

2. (offline) Starting from 2^t arbitrary starting points, iterate the compression function with the fixed message C.

3. (offline) Test each image point x obtained in Step 2 against each of the messages M_j. If a match is found, the state reached after M_j is x.

Complexity analysis. Again, the attack is successful with high probability if $t+u \geq n-s$.

The optimal trade-off is obtained with $u = s$ and $t = n - 2s$, with the following complexity:

Step 1: $2^{s+u/2+n/2} = 2^{n/2+3s/2}$ **Step 2:** $2^{s+t} = 2^{n-s}$ **Step 3:** $2^{t+u} = 2^{n-s}$

The optimal complexity is $\tilde{\mathcal{O}}(2^{4n/5})$ when $s = n/5$. More generally, we obtain an attack with complexity $\tilde{\mathcal{O}}(2^{n-s})$ when $s \leq n/5$.

4.5 Key-recovery attack against HMAC-GOST

Finally, we consider `HMAC` used with a hash function with an internal checksum. The checksum computes the sum (XOR) of all message blocks, and uses this sum as an extra input to the finalization function. This technique is supposed to improve the security of a hash function, assuming that controlling the checksum would be an additional challenge for an adversary. Previous work have shown that the checksum fails to improve the security on many cases [GK08], but we obtain a more surprising result: the checksum makes the hash function *weaker*, enabling key-recovery attack faster than exhaustive search on `HMAC`.

Notable examples of hash functions with a checksum are the standards from the Russian government: GOST [Dol10] and Streebog [DD13]. Figure 4.4 shows the evaluation of `HMAC` with a Merkle-Damgård hash function with a checksum.

State recovery. First, we note that the checksum does not prevent the state-recovery attacks. Concretely, the attack of Section 4.3 is based on detecting collisions between pairs of messages

$$M_1 = \beta^z \,\|\, \alpha \,\|\, \beta^{2^{n/2}+\ell} \qquad\qquad M_2 = \beta^{z+\ell} \,\|\, \alpha \,\|\, \beta^{2^{n/2}}$$

Since the messages have the same checksum, a collision in the state will be preserved.

We use the state-recovery attack to recover the internal state x_* before the checksum is processed.

Key recovery. In HMAC, $k \oplus \texttt{ipad}$ is prepended to a message M, and $(k \oplus \texttt{ipad}) \parallel M$ is hashed by the underlying hash function H. Therefore, the final checksum value is $\sigma = \texttt{Sum}^{\oplus}((k \oplus \texttt{ipad}) \parallel M) = k \oplus \texttt{ipad} \oplus \bigoplus M_i$. We take advantage of the fact that σ contains the key, and can be controlled by changing the message. We inject a known difference in the checksum to perform a kind of related-key attack on the finalization function g, even though we have access to a single HMAC key.

More precisely, we use Joux's multicollision attack [Jou04] to generate a large set of messages with the same value x_* before the finalization, but with different values of the checksum. We detect MAC collisions among those messages, and we assume that the collisions happen when processing the checksum. For each such collision, we have $g(x_*, k \oplus \texttt{ipad} \oplus \bigoplus M_i) = g(x_*, k \oplus \texttt{ipad} \oplus \bigoplus M'_i)$, and we compute the input difference $\Delta M = \bigoplus M_i \oplus \bigoplus M'_i$.

Finally, we compute $g(x_*, m)$ offline for a large set of random values m, and we collect collisions. Again, we compute the input difference $\Delta m = m \oplus m'$ for each collision, and we match Δm to the previously stored ΔM. When a match is found between the differences we look for the corresponding values and we have $k \oplus \texttt{ipad} \oplus \bigoplus M_i = m$ (or m') with high probability. This gives the value of the key k.

Application. We obtain an attack with complexity $2^{3n/4}$ on HMAC-GOST and an attack with complexity $2^{4n/5}$ on HMAC-Streebog. The attack on HMAC-Streebog is less efficient because Streebog uses the HAIFA construction, which makes the key-recovery attack more complex.

4.6 Conclusion

In the chapter we have demonstrated state-recovery attacks for hash-based MACs with complexity below 2^n. In particular, our generic attacks are more efficient than dedicated attacks (presented as distinguishing-H attacks) that have been published against hash functions of the MD-SHA family [KBP+06; RR08; WYW+09].

For Merkle-Damgård hash functions we obtain an attack with complexity $\tilde{\mathcal{O}}(2^{n/2})$, matching the security proof. For HAIFA hash functions, we obtain an attack with complexity $\tilde{\mathcal{O}}(2^{4n/5})$, leaving a gap with the proven security bound.

In the full paper [DL16] (Appendix G), we extend the framework for state-recovery attacks. We observe that our state-recovery attacks are similar to collision-finding algorithms: the attack from Section 4.3 is similar to Pollard's rho algorithm, while the attack from Section 4.4 is similar to the parallel algorithm of van Oorschot and Wiener [vW94]. We point out that Pollard's rho algorithm returns one specific collision with high probability: the root of the giant tree. This property is used in our state-recovery attack. We show that the algorithm of van Oorschot and Wiener also returns collisions from a small set. Based on this observation, we obtain different attacks with various trade-offs between the message length and the complexity of the attacks, as shown in Figure 4.5.

Those techniques have also been used to devise universal forgery attacks, where the attacker is given a long challenge, and must predict the MAC of the challenge after interacting with the oracle [PW14; GPS+14; DL14].

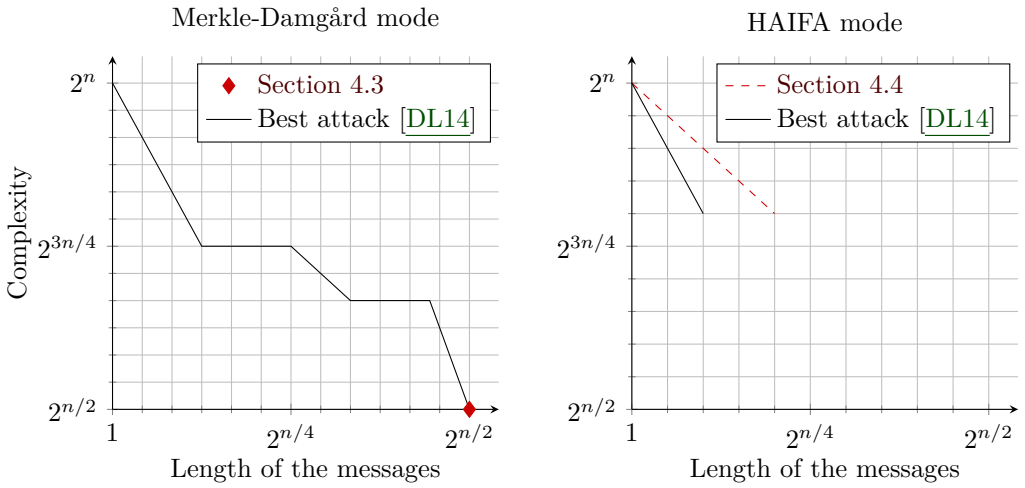


Figure 4.5: Trade-offs between the message length and the complexity for state-recovery attacks.

5. Generic Attacks Against Encryption Modes

This chapter is based on articles [BL16a] and [LS18], found in Appendix H and I.

Block ciphers (such as DES or the AES) are probably the most widely used cryptographic primitives. Formally, a block cipher is just a keyed family of permutations over n-bit blocks, but when combined with a mode of operation, it can provide confidentiality (*e.g.* using CBC, or CTR), authenticity (*e.g.* using CBC-MAC, PMAC, or GMAC), or authenticated encryption (*e.g.* using GCM, CCM, or OCB). A mode of operation defines how to divide a message into blocks, and how to process the blocks one by one with some chaining rule.

When analyzing the security of block cipher modes of operations, there is frequently a term $\sigma^2/2^n$ in the security proof (with σ the total length of the queries made by the adversary, and n the block size), due to the PRF/PRP switching lemma [BKR94]. Indeed, a block cipher is idealized as a pseudo-random *permutation* (PRP) family, but proofs often need it to behave as a pseudo-random *function* (PRF) family. The difference between a PRF and a PRP is that PRFs have collisions, which are detected with probability at most $\sigma^2/2^n$. More generally, most proof techniques fail at the birthday bound; this explains why we use modes with birthday security.

5.1 CBC and CTR

Two important examples are the CBC (Cipher-Block Chaining) mode and the CTR (Counter) mode; they are the most widely used modes of operation used for symmetric encryption.

The CBC mode. CBC is one of the oldest encryption modes, standardized in 1980 [NBS80], and still widely used. The message M is divided into blocks $m_i, 0 \leq i < L$ and is encrypted

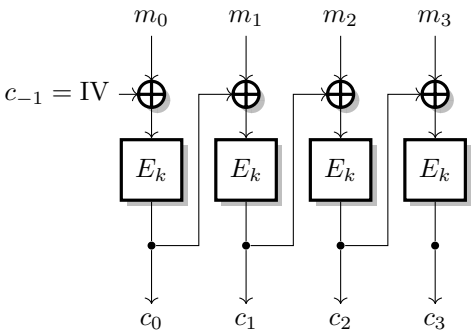


Figure 5.1: CBC mode.

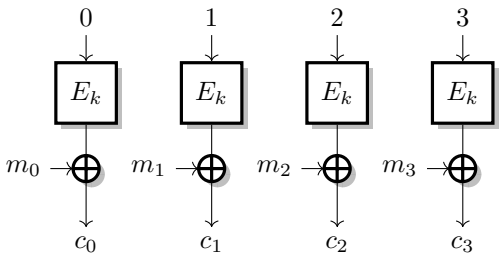


Figure 5.2: CTR mode.

as (see Figure 5.1):

$$c_i = E_k(m_i \oplus c_{i-1}),$$

where c_{-1} is an initialization value usually denoted as IV. For simplicity, we assume that all message sizes are a multiple of the block size.

CBC has been proven secure up to $2^{n/2}$ blocks of messages [BKR94; PR00]. On the other hand there is a simple birthday attack against CBC: after $2^{n/2}$ message blocks encrypted with the same key (in the same message or in different messages), a collision between two ciphertext blocks $c_i = c_j$ is expected. Since E_k is a permutation, a collision in the output means that the inputs are the same ($m_i \oplus c_{i-1} = m_j \oplus c_{j-1}$), which reveals the XOR of two plaintext blocks:

$$m_i \oplus m_j = c_{i-1} \oplus c_{j-1}.$$

The CTR mode. The CTR mode was first proposed in 1979 [DH79], but standardized more recently in 2001 [NIS01]. It is now widely used: in particular, it provides the encryption part of authenticated encryption modes CCM and GCM. The counter mode turns a block cipher into a stream cipher: a counter is encrypted to generate a keystream that is xored with the message. More precisely, the message M is divided into blocks $m_i, 0 \leq i < L$ which are encrypted as (see Figure 5.2):

$$c_i = E_k(i) \oplus m_i,$$

For simplicity we consider a stateful variant of the counter mode with a global counter that is maintained across messages and initialized as 0.

If E_k is a pseudo-random *function*, rather than a pseudo-random *permutation*, then CTR is secure up to 2^n blocks [BDJ+97]. However, in the common case where E_k is a permutation, CTR is proven secure only up to $2^{n/2}$ blocks, and there is a matching distinguishing attack with complexity $2^{n/2}$. If a message where all the plaintext blocks are equal is encrypted, then all the ciphertext blocks will be pairwise different. Indeed, all the $E_k(i)$ are unique, because E_k is a permutation. More generally, with known plaintext, the adversary can reconstruct the keystream blocks $c_i \oplus m_i$ and observe that they are distinct. This distinguishes the counter mode from an ideal encryption scheme, because random data would collide after roughly $2^{n/2}$ blocks.

More interestingly, McGrew proposed a method to eliminate *impossible values* for the plaintext [McG12]. Given two encrypted blocks $c_i = E_k(i) \oplus m_i$ and $c_j = E_k(j) \oplus m_j$, we have $E_k(i) \neq E_k(j)$, therefore we deduce

$$m_i \oplus m_j \neq c_i \oplus c_j.$$

Using prior knowledge about the plaintext (such as a dictionary), an attacker can eventually recover the right plaintext.

Security of CTR and CBC. It is interesting to compare the birthday attacks against CBC and CTR. Both attacks have the same complexity, and show that the proofs are tight. However, they are quite different:

- The CBC attack is based on detecting collisions, and directly leaks information about the plaintext;
- The CTR attack is based on detecting the *absence* of collisions, and only leaks impossible values of the plaintext.

In general, there is a folklore belief that the leakage of the CTR mode is not as bad as the leakage of the CBC mode. For instance, Ferguson, Schneier and Kohno wrote (in the context of a 128-bit block cipher):

> *CTR leaks very little data. [...] It would be reasonable to limit the cipher mode to 2^{60} blocks, which allows you to encrypt 2^{64} bytes but restricts the leakage to a small fraction of a bit. When using CBC mode you should be a bit more restrictive. [...] We suggest limiting CBC encryption to 2^{32} blocks or so.*
>
> — Cryptography Engineering, [FSK11, Section 4.8.2]

Our results. In this chapter we study the impact of birthday attacks on CBC and CTR. We show that both modes suffer plaintext-recovery attacks with birthday complexity in realistic settings. Moreover, we demonstrate the CBC attack in practice with ciphers with 64-bit blocks, showing that they do not offer meaningful security.

5.2 Birthday-bound security

While block ciphers with n-bit blocks and κ-bit keys are required to resist any attack with up to 2^n data and 2^κ time, most modes of operation are only proven secure up to $2^{n/2}$ blocks of plaintext. Rogaway summarizes the situation as follows:

> *In general, unless special efforts are taken, almost any mode of operation based on an n-bit blockcipher will admit attacks that employ $\sigma = 2^{n/2}$ blocks. [...]*
> *Do birthday-bound attacks on CBC, CFB, and OFB actually matter? They are of relatively little concern when the blockcipher has a blocksize of $n = 128$ bits, but the attacks can be a serious concern when employing a blockcipher of $n = 64$ bits, requiring relatively frequent rekeying to keep $\sigma \ll 2^{32}$.*
>
> — Phillip Rogaway, [Rog11, p. 36]

The main countermeasure against birthday attacks is to change the key frequently, before reaching the birthday bound under any single key. For instance, if the key is changed after $2^{n/2-16}$ blocks, a collision attack against a given key only succeeds with probability 2^{-32}, and an attacker has to collect 2^{32} different sessions to mount an attack with high probability.

If implemented properly, this is an effective security measure to strengthen block ciphers with a small block-size. We stress than the key must be changed *well before* $2^{n/2}$ blocks, despite misleading recommendations in published standards. For instance, ISO used to recommend:

> *Generally, for a block cipher with block size of n bits, the maximum amount of plaintext that can be encrypted before rekeying must take place is $2^{(n/2)}$ blocks, due to the birthday paradox. As long as the implementation of a specific block cipher do not exceed these limits, using the block cipher will be safe.*
>
> — ISO/IEC JTC1 SC27, May 2014 [ISO14]

NIST had a similar recommendation for 3DES, a legacy standard with a 64-bit block size (denoted as TDEA in NIST documents):

> *The security of TDEA is affected by the number of blocks processed with one key bundle. One key bundle **shall not** be used to process more than 2^{32} 64-bit data blocks.*
>
> — NIST, January 2012 [NIS12]

Those recommendations are inadequate, because collisions occur with high probability (close to 0.39) with only $2^{n/2}$ blocks. Similarly, many internet standards (IPsec, TLS, SSH)

do not specify a session limit, and many applications allowed very long sessions with the same key, up to or even above $2^{n/2}$ blocks.

On the practical (in-)security of 64-bit block ciphers

With a modern block cipher with 128-bit blocks such as AES, the birthday bound corresponds to 256 EB (2^{68} bytes). While this leaves a safe margin for common uses of cryptography today, the resulting security is still lower than could be expected of 128-bit security and might be an issue in the future. As a point of comparison, the worldwide data storage capacity is estimated to be around 8 ZB (2^{73} bytes) in 2021 [Sta22b], and the global Internet traffic is estimated around 4 ZB (2^{72} bytes) per year in 2022 [Sta22a].

More strikingly, the birthday bound corresponds to only 32 GB for a block cipher with 64-bit blocks. There are real-world scenarios in which this amount of data may be encrypted under a block cipher with 64-bit blocks: 3G telephony connections (UMTS) are encrypted with KASUMI using the `f8` mode, and Internet protocols like OpenVPN, TLS, SSH, and IPsec use 3DES or Blowfish as legacy ciphers for interoperability. In these scenarios, 32 GB of data can be transferred in less than one hour with a fast connection. In the following, we investigate the practical impact of birthday attacks, with a focus on block ciphers with 64-bit blocks.

Previous work. McGrew has investigated plaintext-recovery attacks on 64-bit ciphers in `CBC`, `CFB`, and `CTR` modes [McG12]. He studied the amount of information leakage by assuming that the attacker can observe all the ciphertext in an encrypted stream, and she has some (incomplete) knowledge of the plaintext. We revisit the results of McGrew by considering a stronger adversary model (with chosen plaintext rather than known plaintext), and more advanced algorithms to solve the underlying problems.

5.3 Plaintext-recovery attack against `CBC`

We consider collision attacks against the `CBC` mode in the chosen-prefix secret-suffix model [HRR+15]. The attacker must recover a t-block secret S, given access to an oracle that computes the encryption of $M \parallel S$ for a chosen message M. Using the collision attack above, the attacker queries 2^q arbitrary messages of u block each, corresponding to $\sigma = (t + u) \times 2^q$ blocks of data. She expects on average $\sigma^2 \times 2^{n-1}$ collisions among the ciphertext blocks.

Using short messages. When $u \ll t$, most collisions reveal the XOR between two blocks from S. The attacker collects these relations until she learns the XOR between all pairs of blocks from S. We model a relation as an edge in a graph whose vertexes are the plaintext blocks; with high probability, the graph will be fully connected when there are more than $\Theta(t \ln(t))$ edges. We deduce the following complexity (up to constant factors):

$$\sigma^2 \times 2^{n-1} \approx t \ln(t)$$
$$\sigma \approx \sqrt{t \ln(t)} \times 2^{n/2}$$

At this point, there are $\sigma = \sqrt{t \ln(t)} \times 2^{n/2}$ ciphertext blocks from S, and $u/t \times \sigma = u \times \sqrt{\ln(t)/t} \times 2^{n/2}$ from M. We expect $u \ln(t) > 1$ collisions across the two sets; this reveals at least one block of secret, and the others are deduced using the relations between the blocks from S.

Using longer messages. When $u \gtrsim t$, most collisions are between two blocks from M, and the attacker doesn't learn anything from such collisions. But some collisions reveal the XOR between a block from S and a block from M; we expect $u2^q \times t2^q \times 2^{-n} = tu \times 2^{2q-n}$ collisions of this type. In order to recover the full secret, the attacker needs on average $t \times H_t$ such collisions, following the analysis of the coupon collector's problem (with H_t the harmonic numbers $H_t = \sum_{i=1}^{t} 1/i$). We deduce the complexity of the attack:

$$t \times H_t \approx t \ln(t) = tu \times 2^{2q-n}$$
$$\sigma = u2^q = \sqrt{u \ln(t)} \times 2^{n/2}$$

Using longer queries increases the data complexity of the attack, but it reduces the total number of queries from $\mathcal{O}(\sqrt{\ln(t)/t} \times 2^{n/2})$ to $\mathcal{O}(\sqrt{\ln(t)/u} \times 2^{n/2})$. In practice, this results in a faster attack overall when there is a minimal delay for each query.

With rekeying. If the lifetime of a key is limited to $2^r \lesssim 2^{n/2}$ blocks of data, this increases the complexity of the attack, but does not prevent it. Using the short message attack, we can make $2^r/t$ queries per key, so that we make 2^q queries spread over $t \times 2^{q-r}$ different keys. Collision between ciphertext blocks encrypted with different keys are useless, but we still expect on average 2^{2r-n-1} collisions per key, *i.e.* $t \times 2^{q+r-n-1}$ useful collisions in total. We obtain the following complexity (up to constant factors):

$$t \times 2^{q+r-n-1} \approx t \ln(t)$$
$$\sigma = t \times 2^q \approx t \ln(t) \times 2^{n-r}$$

If r is slightly below $n/2$ the attack still has complexity close to $2^{n/2}$, because for each key there is a non-negligible probability that a collision occurs.

5.3.1 Usage of 64-bit block ciphers

When we performed this analysis in 2016, there was a non-negligible usage of block ciphers with 64-bit blocks, mostly 3DES and Blowfish. In most cases, there was no limit to session length, or a limit close to 2^{32} blocks (insufficient to prevent the attack).

Usage in TLS. In TLS 1.0 and 1.1, 3DES is the mandatory encryption algorithm, so all TLS libraries implemented it and a vast majority of web servers supported it. Before we disclosed our results, mainstream TLS libraries still treated 3DES to be at the same security level as AES. For example, OpenSSL included 3DES ciphersuites in its HIGH-security list. Until 2014, OpenSSL even considered 3DES as more secure than AES-128 because of its larger key.

In 2016, telemetry results showed around 1% usage of 3DES in web browsers for HTTPS traffic. Actually, 3DES usage increased in 2015, after RC4 was deprecated due to practical attacks [ABP+13]. 3DES was recommended as the backup cipher in place of RC4, for legacy server or clients without AES support. Surprisingly, scan results showed that many servers that would use 3DES actually supported AES, but *preferred* to use 3DES.

TLS does not explicitly limit the amount of data encrypted with the same key. When using HTTPS, HTTP/1.1 supports sending multiple messages in the same TLS session. In practice, web browsers did not force any rekeying, and we found many web servers that supported long-lived connections with a large amount of data without rekeying. We measured that 0.6% of the top 100k websites would both use 3DES with an up-to-date browser, and supported long sessions. This included high-profile websites such as `ebay.com`.

Usage in OpenVPN. OpenVPN is a popular open-source VPN solution. Before version 2.4 (released in December 2016), the default encryption for the transport protocol of OpenVPN was Blowfish, a block cipher with 64-bit blocks, in CBC mode. OpenVPN supports two different ways of generating session keys to encrypt the messages. In pre-shared-key mode, static keys are used for all the traffic. In particular, there is no limit to the lifetime of those keys. In TLS mode, session keys are generated with a TLS handshake, using certificates to authenticate the peers. The session keys are updated periodically, with limits on the number of packets, the number of bytes, or a session time. The default configuration rekeys the tunnel every hour. In addition, the tunnel is rekeyed shortly before 2^{32} packets in TLS mode, because the packet counter is a 32-bit integer. In pre-shared-key mode, the packet counter is a 64-bit integer.

Usage in SSH. In SSH version 1, the strongest available ciphers are all block ciphers with 64-bit blocks: 3DES, Blowfish, and IDEA, and although SSH 2 introduced AES, it still labeled 3DES as the mandatory cipher. Consequently, popular implementations such as OpenSSH still used 3DES as the default cipher for SSH-1 connections. SSH standards recommend that implementations rekey after every 1 GB of data (or after every $2^{n/4}$ blocks when $n \geq 128$ [NKB06]), but many popular SSH clients do not implement this feature.

5.3.2 Practical attacks against CBC with 64-bit block ciphers

As explained in Section 1.3, the man-in-the-browser setting is a practical model for attacks against websites, where the attacker targets an authentication token that is included in every query. Therefore, she can directly apply the chosen-prefix secret-suffix attack given above. We observe that most of the traffic generated is known or predictable. The requested URL is chosen by the attacker, and all the headers excepted the authentication token are predictable and can be observed in any plaintext HTTP traffic.

We have demonstrated two versions of the attack, against OpenVPN tunneling and against TLS encryption. We assume that the token takes two blocks, and that the attacker has aligned it to a block boundary using other headers.

Proof-of-concept attack against OpenVPN. To demonstrate the attack against OpenVPN, we used a pre-shared-key tunnel between two physical machines running Linux, with Firefox Developer Edition 47.0a2 on one side, and an nginx server on the other side. Access to the server was protected by BasicAuth (in plaintext inside the tunnel). Using the default OpenVPN settings, the tunnel was encrypted with Blowfish in CBC mode.

We used JavaScript code to send a large number of requests to the server though the tunnel. We found that increasing the size of the request to 4 kB did not significantly reduce the query rate, but reduced the number of queries required for the attack. In our setting, the browser generated about 2900 requests per second, using several web `Workers` running in parallel. The full attack is expected to use about $2^{36.6}$ blocks (780 GB) to recover a two-block secret out of 4 kB messages, corresponding to about 19 hours in this setting. In our demo, it took 18.6 hours and 705 GB, and we successfully recovered the 16-byte authentication token.

Proof-of-concept attack against TLS. In our setup we used Firefox Developer Edition 47.0a2 running on Linux and an IIS 6.0 server in a Windows Server 2003 R2 SP2 Virtual machine. The default configuration of IIS 6.0 with all recommended updates offered only RC4 and 3DES ciphers, and 3DES was used with Firefox and other browsers where RC4 is disabled. Moreover, IIS 6.0 supports HTTP/1.1 and keeps an active connection open for an arbitrary long time.

We used JavaScript to generate queries but we note that with several `Workers` running in parallel, Firefox opened a few parallel connections to the server (typically, 6) and split

the requests over these connections. For our attack, we need to maximize the throughput over a *single* connection. This can be achieved by disturbing some of the connections so that most of the requests are sent in a single connection. In our setup, we used `iptables` rules to limit the rate of all connections except one.

On Firefox Developer Edition 47.0a2, with a few dozen `Workers` running in parallel, we could send up to 2000 requests per second in a single TLS connection. To further reduce the time needed for the attack, we injected a padding cookie to expand the requests to 4 KB (512 blocks). In our setting this reduced the rate to 1500 requests per second, but still lead to a faster attack. Again, the full attack should require $2^{36.6}$ blocks (780 GB) to recover a two-block token, which should take 38 hours in our setting. Experimentally, we recovered the token (a two-block cookie) after only 30.5 hours and 610 GB.

5.3.3 Responsible disclosure.

Our attack was responsibly disclosed in 2016 under the name Sweet32, using the website `sweet32.info`. We have contacted the OpenVPN team, as well as various website owners, browser vendors, and TLS libraries. The TLS vulnerability received CVE number `CVE-2016-2183`, and the OpenVPN vulnerability is tracked as `CVE-2016-6329`.

Several standards have been updated to deprecate 64-bit block ciphers or impose data limits:

- NIST has reduced the maximum amount of plaintext allowed to be encrypted under a single 3DES key from 2^{32} blocks to 2^{20} [NIS17]. They later announced that 3DES usage for encryption would be deprecated in 2023, and disallowed after 2023 [NIS19].
- ISO has updated its recommendations on block size [ISO19];

Software vendors have also deprecated 3DES:

- Mozilla has implemented data limits (at most 2^{20} records for 3DES) in Firefox 51 (released in January 2017) and disabled 3DES by default in Firefox 93 (released in October 2021). Google disabled 3DES in Chrome 93 in August 2021.
- OpenVPN 2.4 (released in December 2016) has introduced ciphersuite negotiation, defaulting to AES-GCM, instead of a fixed ciphersuite defaulting to Blowfish-CBC. Moreover, it rekeys every 2^{23} blocks (64 MB) when using a 64-bit block cipher.
- OpenSSL has deprecated 3DES with version 1.1.0 in August 2016.
- OpenSSH removed 3DES from the default client configuration with version 7.4 in December 2016.

5.4 The missing difference problem

We now consider the security of the `CTR` mode against birthday attacks. There is a well-known distinguisher based on the absence of collision in the keystream, and McGrew has formulated a plaintext-recovery attack as a simple algorithmic problem [McG12]. In the chosen-prefix secret-suffix model (with a single block secret S), an attacker collects two types of ciphertexts blocks $c_i = E_k(i) \oplus m_i$:

- encrypted blocks corresponding to a known plaintext block m_i; we denote the corresponding keystream as $a_i = m_i \oplus c_i = E_k(i)$;
- encrypted blocks corresponding to the fixed secret S; we denote the corresponding ciphertext as $b_j = S \oplus E_k(j)$.

Algorithm 1 Known-prefix sieving.

Input: n, $z < n$, $\mathcal{A}, \mathcal{B}$, $\mathcal{S} \subseteq \{0\}^z \times \{0,1\}^{n-z}$
  $h \leftarrow$ Empty hash table.
  for b **in** $\mathcal{B}$ **do**
    $h[b_{[0\ldots(z-1)]}] \xleftarrow{\cup} \{b_{[z\ldots(n-1)]}\}$
  end for
  for a **in** $\mathcal{A}$ **do**
    $v_a \leftarrow a_{[z\ldots(n-1)]}$
    for v_b **in** $h[a_{[0\ldots(z-1)]}]$ **do**
      Remove $0 \,\|\, (v_a \oplus v_b)$ from $\mathcal{S}$;
    end for
  end for
  return $\mathcal{S}$

McGrew observes that all values $a_i \oplus b_j$ are different from S:

$$i \neq j \;\Rightarrow\; E_k(i) \neq E_k(j) \;\Rightarrow\; a_i \neq b_j \oplus S \;\Rightarrow\; S \neq a_i \oplus b_j.$$

Therefore, we consider an attacker that collects a set of keystream blocks $\mathcal{A} = \{a_i\} \subseteq \{0,1\}^n$ and a set of encryption of the secret $\mathcal{B} = \{b_j\} \subseteq \{0,1\}^n$. We further assume some prior knowledge about the secret: $S \in \mathcal{S} \subseteq \{0,1\}^n$. In order to recover the secret S, the attacker has to solve the following algorithmic problem:

Definition 1 (Missing Difference Problem)**.** Given two sets $\mathcal{A}$ and $\mathcal{B}$, and a hint $\mathcal{S}$, find the value $S \in \mathcal{S}$ such that:

$$\forall (a,b) \in \mathcal{A} \times \mathcal{B},\; S \neq a \oplus b\,.$$

In the information theoretic setting, this problem can be solved with $|\mathcal{A}| = |\mathcal{B}| = \tilde{\mathcal{O}}(2^{n/2})$ for any set $\mathcal{S}$, and requires at least $|\mathcal{A}| \times |\mathcal{B}| = \Omega(2^n)$ when $|\mathcal{S}| \geq 2$. However, the analysis is more complex when taking into account the cost of the computations required to recover S. McGrew introduces two algorithms for this problem:

- A sieving algorithm that iterates over $\mathcal{A} \times \mathcal{B}$ and discards values from $\mathcal{S}$ until there is only one value left. This requires $|\mathcal{A}| = |\mathcal{B}| = \tilde{\mathcal{O}}(2^{n/2})$ and time $\tilde{\mathcal{O}}(2^n)$.
- A searching algorithm that iterates first over $s \in \mathcal{S}$, then over $a \in \mathcal{A}$ and checks whether $s \oplus a \in \mathcal{B}$ to discard s. With an optimal parameter choice, this requires $|\mathcal{A}| = \tilde{\mathcal{O}}(2^{n/2}/\sqrt{|\mathcal{S}|})$, $|\mathcal{B}| = \tilde{\mathcal{O}}(2^{n/2}\sqrt{|\mathcal{S}|})$ and time $\tilde{\mathcal{O}}(2^{n/2}\sqrt{|\mathcal{S}|})$.

In the following we describe more efficient algorithms for this problem. We consider that $\mathcal{A}$ and $\mathcal{B}$ correspond to oracle queries, so that we want to minimize $|\mathcal{A}|$ and $|\mathcal{B}|$ as well as the time complexity.

5.4.1 Known-prefix sieving

In many concrete attack scenarios, an attacker knows some bits of the secret message in advance. We model this by assuming that the secret is known to start with z zero bits.[1] We propose an algorithm to solve the missing difference problem efficiently in this context (Algorithm 1).

Since S is known to start with z zeroes, we look for z-bit collisions between $\mathcal{A}$ and $\mathcal{B}$. With $|\mathcal{A}| = |\mathcal{B}| = \sqrt{n-z} \times 2^{n/2}$ we expect $(n-z) \times 2^{n-z}$ collisions. For each collision,

[1] Our result can be generalized to a known prefix instead of zero prefix, and to $\mathcal{S}$ being an affine space of dimension $n-1$.

Algorithm 2 Fast-convolution sieving.

Input: $\mathcal{A}, \mathcal{B}, t \leq n$
  $C_{\mathcal{A}}, C_{\mathcal{B}}, C_{\mathcal{A}\oplus\mathcal{B}} \leftarrow$ arrays of 2^{n-t} integers initialized to 0;
  for a **in** $\mathcal{A}$ **do**
    Increment $C_{\mathcal{A}}[a_{0..(n-t-1)}]$
  end for
  for b **in** $\mathcal{B}$ **do**
    Increment $C_{\mathcal{B}}[b_{0..(n-t-1)}]$
  end for
  $C_{\mathcal{A}\oplus\mathcal{B}} \leftarrow$ FastConvolution$(C_{\mathcal{A}}, C_{\mathcal{B}})$
  $u \leftarrow \text{argmin}_i\, C_{\mathcal{A}\oplus\mathcal{B}}[i]$
  Run known prefix sieving (Algorithm 1), knowing that $T(S) = u$

we discard the corresponding value from $\mathcal{S}$. Following the analysis of the coupon collector, we expect to discard all wrong values. Therefore, we recover S with $\tilde{\mathcal{O}}(2^{n/2})$ queries and memory, and $\tilde{\mathcal{O}}(2^{n-z} + 2^{n/2})$ operations.

With $z = 0$, the algorithm is the same as McGrew's sieving algorithm, but with $z \geq n/2$ we have an algorithm with optimal complexity $\tilde{\mathcal{O}}(2^{n/2})$, more efficient than McGrew's searching algorithm.

5.4.2 Fast-convolution sieving

When we have no prior information on the secret ($\mathcal{S} = \{0,1\}^n$), we can reduce the complexity of the sieving algorithm by using sets $\mathcal{A}$ and $\mathcal{B}$ of size $2^m \gg 2^{n/2}$, rather than $\tilde{\mathcal{O}}(2^{n/2})$ as required to uniquely identify S. If we consider all the values $a \oplus b$ for (a, b) in $\mathcal{A} \times \mathcal{B}$, we expect that they are close to uniformly distributed over $\{0,1\}^n \setminus S$, so that every value except S is reached about 2^{2m-n} times, while S is never hit. Increasing m makes the gap more visible than with sets of size only $\tilde{\mathcal{O}}(2^{n/2})$. Therefore, we consider buckets of several candidates s, and accumulate the number of $a \oplus b$ in each bucket. With buckets of 2^t values, each bucket receives $2^t \times 2^{2m-n}$ values on average, but the bucket containing S receives only $(2^t - 1) \times 2^{2m-n}$ values. If we model the counters as random variables following a binomial distribution, the variance σ^2 is about $2^{m+t/2-n/2}$. Therefore, the bias will be detectable when $\sigma \ll 2^{2m-n}$, *i.e.* when $t < 2m - n$.

Concretely, we use a truncation function T that keeps only $n - t$ bits of an n-bit word. We consider the values $T(a \oplus b)$ for all $(a, b) \in \mathcal{A} \times \mathcal{B}$, and count how many times each value is reached. If m is large enough, the lowest counter corresponds to $T(S)$.

For a given multi-set $\mathcal{X}$, we consider an array of 2^{n-t} counters $C_{\mathcal{X}}$, to represent how many times each value $T(x)$ is reached:

$$C_{\mathcal{X}}[i] = \big|\{x \in \mathcal{X} : T(x) = i\}\big|, \quad i \in \{0, \ldots, 2^{n-t} - 1\}.$$

Our goal is to compute $C_{\mathcal{A}\oplus\mathcal{B}}$ efficiently from $\mathcal{A}$ and $\mathcal{B}$, where $\mathcal{A} \oplus \mathcal{B}$ is the multi-set $\{a \oplus b : (a, b) \in \mathcal{A} \times \mathcal{B}\}$. We observe that:

$$\begin{aligned} C_{\mathcal{A}\oplus\mathcal{B}}[i] &= |\{(a, b) \in \mathcal{A} \times \mathcal{B} : T(a \oplus b) = i\}| \\ &= \sum_{j \in \{0,1\}^{n-t}} |\{a \in \mathcal{A} : T(a) = j\}| \times |\{b \in \mathcal{B} : T(b) = i \oplus j\}| \\ &= \sum_{j \in \{0,1\}^{n-t}} C_{\mathcal{A}}[j] C_{\mathcal{B}}[i \oplus j] \end{aligned}$$

This is a form of convolution that can be computed efficiently using the Fast Walsh-Hadamard Transform, in the same way we use the Fast Fourier Transform to compute

circular convolutions. We obtain all the values $C_{\mathcal{A}\oplus\mathcal{B}}[i]$ simultaneously with complexity $\tilde{\mathcal{O}}(2^{n-t})$.

This results in an algorithm with complexity $\tilde{\mathcal{O}}(2^m + 2^{n-t})$ (Algorithm 2), with $m > (n+t)/2$. We optimize the complexity by choosing $t = n/3$ and $m = 2n/3 + \ln(n)/2$ such that the time complexity, data complexity, and memory usage are all in $\tilde{\mathcal{O}}(2^{2n/3})$.

5.5 Plaintext-recovery attack against CTR

The missing difference algorithms give new attacks against the CTR mode and other related constructions, such as Wegman-Carter-Shoup MACs.

5.5.1 Attacking CTR in the chosen-prefix secret-suffix model

We consider collision attacks against the CTR mode in the chosen-prefix secret-suffix model [HRR+15], given an oracle $\mathcal{O} : M \mapsto \mathcal{E}(M \parallel S)$.

Instead of having the chosen message M and the secret S encrypted in separate blocks, the attacker can use a message M of length $n/2$, so that the first block of plaintext contains $n/2$ known bytes and $n/2$ secret bytes. By making a large number of queries with the same message M, she applies the known prefix sieving algorithm to this block and recovers the first $n/2$ secret bytes with complexity $\tilde{\mathcal{O}}(2^{n/2})$.

Then she makes a set of queries with an empty message M, so that the first block of plaintext contains n bytes of S. Since the first $n/2$ bytes have been recovered already, she applies the known-prefix sieving again with complexity $\tilde{\mathcal{O}}(2^{n/2)}$. Eventually, the full secret is recovered iteratively with complexity $\tilde{\mathcal{O}}(|S| \times 2^{n/2})$.

In practice, the man-in-the-browser setting gives enough freedom to the attacker to do this block splitting by varying the length of some headers.

5.5.2 Attacking Wegman-Carter-Shoup MACs

Because the fast convolution algorithm requires few assumptions, it is applicable to other modes of operation based on CTR. We consider the Wegman-Carter-Shoup MAC construction [Sho96]; it uses a block cipher E and a keyed universal hash function h, with keys k_1 and k_2. Given a message M and a nonce N, the MAC is defined as:

$$\texttt{MAC}(N, M) = h_{k_1}(M) \oplus E_{k_2}(N)$$

To apply our attack, we use two fixed messages M and M', and we capture many values $\texttt{MAC}(N, M)$ in a list $\mathcal{A}$ and values $\texttt{MAC}(N', M')$ in a list $\mathcal{B}$, all using unique nonces. We remark that $\forall N \neq N' : E_{k_2}(N) \oplus E_{k_2}(N') \neq 0$, therefore we recover $h_{k_1}(M) \oplus h_{k_1}(M')$ by solving the missing difference problem with $\mathcal{A}$ and $\mathcal{B}$.

In practical constructions such as GMAC [MV04] and Poly1305 [Ber05], h is a polynomial evaluation and recovering $h_{k_1}(M) \oplus h_{k_1}(M')$ is sufficient to recover the hash key k_1, leading to universal forgeries.

Comparison with previous attacks against GMAC. There are several known attacks against GCM and GMAC, but none of them seems to allow universal forgery with just $2^{2n/3}$ blocks of data and $2^{2n/3}$ computations. In particular, Handschuh and Preneel [HP08] gave a weak-key attack that can also be used to recover the hash key without weak key assumptions, using roughly $2^{n/2}$ messages of $2^{n/2}$ blocks. Later work extended these weak key properties [Saa12; PC15] but an attack still requires about 2^n blocks in total when no assumptions are made about the key.

A work done in parallel with ours recovers the hash key using $\mathcal{O}(2^{n/2})$ short queries, and $\mathcal{O}(2^n)$ computation [LP18]. Actually, their attack is essentially equivalent to the

missing difference attack, using the basic sieving algorithm instead of the fast-convolution sieving.

5.6 Conclusion

The missing difference attacks against CTR of Section 5.5 and the collision attacks against CBC of Section 5.3 are two different ways block cipher modes can fail beyond the birthday bound. They exploit different properties of the modes, but result in similar attacks in terms of concrete requirements. This contradicts previous beliefs that CTR is less sensitive to birthday attacks than CBC. These techniques can be used against many other modes of operations (OFB, CFB, ...); most of them will be vulnerable to at least one the attacks, unless they have been carefully designed to provide security beyond the birthday bound.

Plaintext recovery for OFB. It is interesting to consider the security of OFB in light of the previous results. OFB turns a block cipher into stream cipher by iteratively encrypting an IV:

$$z_{-1} = \mathrm{IV} \qquad z_i = E_k(z_{i-1}) \qquad c_i = z_i \oplus m_i$$

If OFB is used with many short messages with different IVs, then collisions in the sequence z_i are expected. A collision can be detected with partial knowledge of the plaintext; it implies that the next z_i also collide and this leaks plaintext information.

On the other hand, if OFB is used with a single long message (or in a stateful variant without IV), then the z_i sequence follows a cycle of E_k. With high probability, the cycle has length close to 2^n, and no collision is expected around the birthday bound. In this case, the missing difference attack recovers plaintext data.

Plaintext recovery for f8. The f8 mode used in UMTS telephony is a variant of the OFB and CTR modes, that was apparently designed to avoid the collision attack against OFB:

$$A = E_{k'}(\mathrm{IV}) \qquad z_{-1} = 0 \qquad z_i = E_k(A \oplus i \oplus z_{i-1}) \qquad c_i = z_i \oplus m_i$$

Our attack can be applied to the first block of ciphertext. Indeed, the first block of message i is encrypted as $c_0 = m_0 \oplus E_k(E_{k'}(\mathrm{IV}))$, where the value $E_k(E_{k'}(\mathrm{IV}))$ is unique for all the messages encrypted with a given key.

Practical demonstration. When the block-size n is equal to 64 bits (or smaller), a plaintext-recovery attack can be carried out in practice in a realistic setting, with a man-in-the-browser recovering an authentication token. This should not come as a surprise to cryptographers, but practitioners tended to downplay the impact of these attacks, and ciphers with 64-bit block sizes were still in use in many protocols when we disclosed our results. By implementing an attack against a popular protocol under real-world conditions, we managed to convince them to deprecate those ciphers. Since legacy block ciphers are often used with CBC but very rarely with CTR, we only demonstrated the CTR attack in practice.

6. Generic Attacks Against Beyond-Birthday-Bound MACs

This chapter is based on article [LNS18], found in Appendix J.

In order to avoid birthday attacks, several block-cipher based MACs have been proposed with an internal state larger than the primitive size. In particular, several constructions use a $2n$-bit internal state so that collisions in the state are only expected after 2^n queries.

6.1 Double-block hash-then-sum MACs

Several block-cipher based MACs with security beyond the birthday bound have been proposed in the last years. Yasuda first proposed `SUM-ECBC` [Yas10], a beyond birthday bound (BBB) secure deterministic MAC with a proof of security up to $2^{2n/3}$ queries. However, this construction has rate 1/2 and Yasuda later proposed `PMAC+` [Yas11], a BBB secure MAC achieving rate 1. This inspired several other constructions like `3kf9` [ZWS+12], `LightMAC+` [Nai17], `GCM-SIV2` [IM16], and single key `PMAC+` [DDN+17].

Interestingly, all the above designs share a common structure: the $2n$-bit internal state is divided in two n-bit parts, that we denote Σ and Θ, and the final MAC is computed as:

$$\texttt{MAC}(M) = E\big(\Sigma(M)\big) \oplus E'\big(\Theta(M)\big),$$

where E and E' denote the block cipher with potentially different keys. The functions Σ and Θ can be seen as two n-bit universal hash functions computed on the message. This structure has been called double-block hash-then-sum, and it will be the focus of this chapter. All these MACs have a security proof up to $2^{2n/3}$ queries [DDN+18], but there are no known attacks with less than 2^n queries.

Our results. We describe a new cryptanalysis technique for double-block hash-then-sum MACs, that generalizes the collision attack of Preneel and van Oorschot. Instead of looking for a pair of messages so that the full state collides, we look for a quadruple of messages, which can be seen either as two pairs colliding on the first half of the state, or two pairs colliding on the second half. Since the finalization function combines the halves with a sum, we can detect such a quadruple because the corresponding MACs sum to zero, and can usually amplify this filtering. Moreover, when the messages are well constructed, the relations defining the four collisions create a linear system of rank only three, so that we expect one good quadruple out of 2^{3n}. Therefore, we only need four lists of $2^{3n/4}$ queries, and we expect one good quadruple out of the 2^{3n} choices in the four lists.

The attacks have a complexity of $2^{3n/4}$ in the information theoretic model (the model used for most MAC security proofs), but we note that an attacker needs more than 2^n operations to create a forgery. However, we have found a variant of our attack against `SUM-ECBC` and `GCM-SIV2` with total complexity below 2^n, using $\mathcal{O}(2^{6n/7})$ queries and

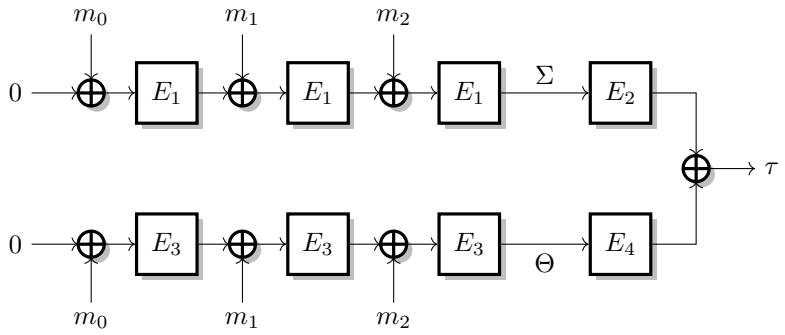


Figure 6.1: `SUM-ECBC`.

$\tilde{\mathcal{O}}(2^{6n/7})$ operations. As far as we know, this is the first attack with complexity below 2^n against a deterministic beyond-birthday-bound-secure MAC.

After the publication of our results, a more advanced security proof by Kim, Lee, and Lee [KLL20], proved security up to $2^{3n/4}$ queries, achieving tight security.

6.2 Attacking `SUM-ECBC`

We focus on `SUM-ECBC` [Yas10], one the simplest double-block hash-then-sum constructions. We describe a universal forgery attack with $\mathcal{O}(2^{3n/4})$ queries and $\tilde{\mathcal{O}}(2^{3n/2})$ operations, and a variant with total complexity below 2^n, with $\mathcal{O}(2^{6n/7})$ queries and $\tilde{\mathcal{O}}(2^{6n/7})$ operations.

`SUM-ECBC` was designed by Yasuda in 2010 [Yas10], inspired by MAC constructions summing two CBC-MACs in the ISO 9797-1 standard. The scheme uses a block cipher keyed with four independent keys, denoted as E_1, E_2, E_3, E_4. The message M is first divided into n-bit blocks $M = m_0 \,\|\, m_1 \,\|\, m_{L-1}$, and the construction is defined as follows (see Figure 6.1):

$$\begin{aligned} \Sigma(M) &= \sigma_L & \sigma_0 &= 0 & \sigma_{i+1} &= E_1(\sigma_i \oplus m_i) \\ \Theta(M) &= \theta_L & \theta_0 &= 0 & \theta_{i+1} &= E_3(\theta_i \oplus m_i) \\ \texttt{MAC}(M) &= E_2(\Sigma(M)) \oplus E_4(\Theta(M)) \end{aligned}$$

6.2.1 Using Quadruples

Our strategy consists in looking for a quadruple of messages (X, Y, Z, T) such that pairs of values collide for one half of the state. More precisely, we look for quadruples satisfying a relation $\mathcal{R}(X, Y, Z, T)$ defined as:

$$\mathcal{R}(X,Y,Z,T) := \begin{cases} \Sigma(X) = \Sigma(Y) \\ \Theta(Y) = \Theta(Z) \\ \Sigma(Z) = \Sigma(T) \\ \Theta(T) = \Theta(X) \end{cases}$$

In particular, since the MAC is computed as $\texttt{MAC}(M) = E\big(\Sigma(M)\big) \oplus E'\big(\Theta(M)\big)$, it follows that:

$$\mathcal{R}(X,Y,Z,T) \implies \texttt{MAC}(X) \oplus \texttt{MAC}(Y) \oplus \texttt{MAC}(Z) \oplus \texttt{MAC}(T) = 0. \tag{6.1}$$

We build the quadruples with two message injection functions:

$$\begin{aligned} \phi(i) &= 0 \,\|\, i & \psi(i) &= 1 \,\|\, i \\ X = \phi(x) &= 0 \,\|\, x & Y = \psi(y) &= 1 \,\|\, y \\ Z = \phi(z) &= 0 \,\|\, z & T = \psi(t) &= 1 \,\|\, t, \end{aligned}$$

With this construction, the pairs (X, Y), (Y, Z), (Z, T) and (T, X) that we consider always contain a message built with ϕ and message built with ψ. Therefore, we will have the required collisions in Σ or Θ if the difference introduced in the half-state by the second block cancels the difference found after processing the first block.

More precisely, we have

$$\begin{aligned}\texttt{MAC}(\phi(i)) &= E_2\Big(\underbrace{E_1\big(i \oplus E_1(0)\big)}_{\Sigma_0(i)}\Big) \oplus E_4\Big(\underbrace{E_3\big(i \oplus E_3(0)\big)}_{\Theta_0(i)}\Big)\\ \texttt{MAC}(\psi(i)) &= E_2\Big(\underbrace{E_1\big(i \oplus E_1(1)\big)}_{\Sigma_1(i)}\Big) \oplus E_4\Big(\underbrace{E_3\big(i \oplus E_3(1)\big)}_{\Theta_1(i)}\Big)\end{aligned}$$

and the relation $\mathcal{R}$ becomes:

$$\mathcal{R}(x,y,z,t) \Leftrightarrow \begin{cases} x \oplus E_1(0) = y \oplus E_1(1) \\ z \oplus E_3(0) = y \oplus E_3(1) \\ z \oplus E_1(0) = t \oplus E_1(1) \\ x \oplus E_3(0) = t \oplus E_3(1) \end{cases} \Leftrightarrow \begin{cases} x \oplus y \oplus z \oplus t = 0 \\ x \oplus y = E_1(0) \oplus E_1(1) \\ x \oplus t = E_3(0) \oplus E_3(1) \end{cases}$$

Because of the choice of the injection functions, $\mathcal{R}$ reduces to a linear system of rank only three. Therefore, we expect to find one quadruple satisfying the relation out of 2^{3n}, and we can construct 2^{3n} quadruples with just $4 \times 2^{3n/4}$ queries. This gives an attack with data complexity $\mathcal{O}(2^{3n/4})$.

6.2.2 Detecting Quadruples

To complete the attack we need to identify a quadruple satisfying $\mathcal{R}$. We directly observe when $x \oplus y \oplus z \oplus t = 0$, and we also detect the relation on the sum of the MACs following Equation (6.1):

$$\mathcal{R}(x,y,z,t) \Rightarrow \texttt{MAC}(\phi(x)) \oplus \texttt{MAC}(\psi(y)) \oplus \texttt{MAC}(\phi(z)) \oplus \texttt{MAC}(\psi(t)) = 0$$

However, we need a $3n$-bit condition to identify a good quadruple with high probability. We use related quadruples that satisfy $\mathcal{R}$ simultaneously to amplify the filtering. We observe that $\mathcal{R}(x,y,z,t)$ is satisfied if and only if $\mathcal{R}(x \oplus c, y \oplus c, z \oplus c, t \oplus c)$ is satisfied for any constant c.

$$\mathcal{R}(x,y,z,t) \iff \forall c,\ \mathcal{R}(x \oplus c, y \oplus c, z \oplus c, t \oplus c) \tag{6.2}$$

This leads to an attack with $\mathcal{O}(2^{3n/4})$ queries: we consider four sets $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{T}$ of $2^{3n/4}$ values, and we look for a quadruple $(x,y,z,t) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{Z} \times \mathcal{T}$ with:

$$\begin{cases} x \oplus y \oplus z \oplus t = 0 \\ \texttt{MAC}(\phi(x)) \oplus \texttt{MAC}(\psi(y)) \oplus \texttt{MAC}(\phi(z)) \oplus \texttt{MAC}(\psi(t)) = 0 \\ \texttt{MAC}(\phi(x \oplus 1)) \oplus \texttt{MAC}(\psi(y \oplus 1)) \oplus \texttt{MAC}(\phi(z \oplus 1)) \oplus \texttt{MAC}(\psi(t \oplus 1)) = 0. \end{cases} \tag{6.3}$$

In practice, we build consider four lists of size $2^{3n/4}$ to express the search for a quadruple as an instance of the 4-sum problem:

$$\begin{aligned} L_1 &= \{x \,\|\, \texttt{MAC}(\phi(x)) \,\|\, \texttt{MAC}(\phi(x \oplus 1)) : x \in \mathcal{X}\} \\ L_2 &= \{y \,\|\, \texttt{MAC}(\psi(y)) \,\|\, \texttt{MAC}(\psi(y \oplus 1)) : y \in \mathcal{Y}\} \\ L_3 &= \{z \,\|\, \texttt{MAC}(\phi(z)) \,\|\, \texttt{MAC}(\phi(z \oplus 1)) : z \in \mathcal{Z}\} \\ L_4 &= \{t \,\|\, \texttt{MAC}(\psi(t)) \,\|\, \texttt{MAC}(\psi(t \oplus 1)) : t \in \mathcal{T}\} \end{aligned}$$

A 4-sum algorithm returns $(x, y, z, t) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{Z} \times \mathcal{T}$ such that $L_1[x] \oplus L_2[y] \oplus L_3[z] \oplus L_4[t] = 0$. A naive algorithm would take time 2^{3n} to examine all quadruples, but there is a simple improvement with time $\tilde{\mathcal{O}}(2^{3n/2})$: the attacker first builds $L_1 \times L_2$ and $L_3 \times L_4$, then she looks for a collision between the two lists. After finding a collision, we verify that it is not a false positive by testing the relation for another value c.

Universal Forgeries. This attack can be extended to a universal forgery. Indeed, the fixed prefix 0 and 1 can be replaced by v and $v \oplus 1$ for any block v, and when we identify a right quadruple (x, y, z, t) we deduce the value $\Delta_1 = E_1(v) \oplus E_1(v \oplus 1)$ and $\Delta_3 = E_3(v) \oplus E_3(v \oplus 1)$. There is also a length extension property: if (x, y, z, t) is a right quadruple, then $\mathtt{MAC}(v \parallel x \parallel s) \oplus \mathtt{MAC}(v \oplus 1 \parallel y \parallel s) \oplus \mathtt{MAC}(v \parallel z \parallel s) \oplus \mathtt{MAC}(v \oplus 1 \parallel t \parallel s) = 0$ for any suffix s.

Therefore, if we want to forge a MAC for any message m of size $L \geq 2$ blocks we parse it as $m = v \parallel w \parallel s$ (where s has zero, one, or several blocks) and perform the attack to recover Δ_1 and Δ_3. Then we can forge using the previous relation, and Equation (6.2):

$$\begin{aligned}\mathtt{MAC}(v \parallel w \parallel s) = \mathtt{MAC}(v \oplus 1 \parallel w \oplus \Delta_1 \parallel s) \oplus \mathtt{MAC}(v \parallel w \oplus \Delta_3 \parallel s)\\ \oplus \mathtt{MAC}(v \oplus 1 \parallel w \oplus \Delta_1 \oplus \Delta_3 \parallel s)\end{aligned}$$

6.2.3 Optimizing the time complexity.

Equation (6.2) can also be used to reduce the time complexity below 2^n, at the cost of more oracle queries. Indeed, if we consider a subset $\mathcal{C}$ of $\{0,1\}^n$, we have:

$$\begin{aligned}&\mathcal{R}(x, y, z, t)\\ &\quad \Leftrightarrow \forall c \in \mathcal{C}, \mathcal{R}(x \oplus c, y \oplus c, z \oplus c, t \oplus c)\\ &\quad \Rightarrow \forall c \in \mathcal{C},\ \mathtt{MAC}(\phi(x \oplus c)) \oplus \mathtt{MAC}(\psi(y \oplus c)) \oplus \mathtt{MAC}(\phi(z \oplus c)) \oplus \mathtt{MAC}(\psi(t \oplus c)) = 0\\ &\quad \Rightarrow \bigoplus_{c \in \mathcal{C}} \mathtt{MAC}(\phi(x \oplus c)) \oplus \bigoplus_{c \in \mathcal{C}} \mathtt{MAC}(\psi(y \oplus c)) \oplus \bigoplus_{c \in \mathcal{C}} \mathtt{MAC}(\phi(z \oplus c)) \oplus \bigoplus_{c \in \mathcal{C}} \mathtt{MAC}(\psi(t \oplus c)) = 0\end{aligned} \tag{6.4}$$

If we select $\mathcal{C}$ as a linear subspace, then the last expression does not depend on the full (x, y, z, t), but only on their projection on the orthogonal complement of $\mathcal{C}$. Concretely, we use $\mathcal{C} = \{x : x_{[3n/7:n]} = 0\} = \{x : x < 2^{3n/7}\}$, so that the value $\bigoplus_{c \in \mathcal{C}} \mathtt{MAC}(\phi(x \oplus c))$ depends on only $4n/7$ bits of x.

Therefore, we consider the rewritten MAC function

$$\mathtt{MAC}'(v \parallel w) = \textstyle\bigoplus_{c \in \mathcal{C}} \mathtt{MAC}(v \parallel w \oplus c),$$

the following message injections, with a $4n/7$-bit input

$$\phi'(i) = 0 \parallel i|0 \qquad\qquad \psi'(i) = 1 \parallel i|0,$$

and a reduced relation over $4n/7$-bit values:

$$\mathcal{R}'(x, y, z, t) := \begin{cases} x \oplus y = (E_1(0) \oplus E_1(1))_{[3n/7:n]} \\ y \oplus z = (E_3(0) \oplus E_3(1))_{[3n/7:n]} \\ z \oplus t = (E_1(0) \oplus E_1(1))_{[3n/7:n]} \\ t \oplus x = (E_3(0) \oplus E_3(1))_{[3n/7:n]} \end{cases} \Leftrightarrow \begin{cases} x \oplus y \oplus z \oplus t = 0 \\ x \oplus y = (E_1(0) \oplus E_1(1))_{[3n/7:n]} \\ x \oplus t = (E_3(0) \oplus E_3(1))_{[3n/7:n]} \end{cases}$$

Thanks to Equation 6.4, we still have:

$$\mathcal{R}'(x, y, z, t) \Rightarrow \mathtt{MAC}'(\phi'(x)) \oplus \mathtt{MAC}'(\psi'(y)) \oplus \mathtt{MAC}'(\phi'(z)) \oplus \mathtt{MAC}'(\psi'(t)) = 0$$

Since the relation $\mathcal{R}'$ is now only a $12n/7$-bit condition, we use shorter lists, with just $2^{3n/7}$ elements. We also increase the filtering using the same trick as previously. Using the 4-sum algorithm, we locate a right quadruple using $\tilde{\mathcal{O}}(2^{6n/7})$ operations. The attack uses $\mathcal{O}(2^{6n/7})$ queries to generate the lists, because each element is the sum of $2^{3n/7}$ queries.

Table 6.1: Summary of the security for studied modes and our main results. q is the number of queries, ℓ is maximum size of a query, σ is the total number of processed blocks. The expected lower bound and attack complexity is in number of constant length queries ($\ell = \mathcal{O}(1)$). We use "U" for universal forgeries, and "E" for existential forgeries.
Proofs marked with * are invalidated by our results.

Mode	Provable security bounds		Our attacks		
	Advantage	Queries	Queries	Time	Type
`SUM-ECBC` [Yas10]	$\mathcal{O}(\frac{q^3\ell^3}{2^{2n}})$ with $\ell \leq 2^{2n/5}$	$\Omega(2^{2n/3})$	$\mathcal{O}(2^{3n/4})$	$\tilde{\mathcal{O}}(2^{3n/2})$	U
			$\mathcal{O}(2^{6n/7})$	$\tilde{\mathcal{O}}(2^{6n/7})$	U
[KLL20]	$\mathcal{O}(\frac{q^{4/3}\ell^{o(1)}}{2^n} + \frac{q^{4/3}\ell^4}{2^{2n}})$	$\Omega(2^{3n/4})$			
`GCM-SIV2` [IM16]	$\mathcal{O}(\frac{q\ell}{2^n} + \frac{q^3\ell^2}{2^{2n}})$	$\Omega(2^{2n/3})$	$\mathcal{O}(2^{3n/4})$	$\tilde{\mathcal{O}}(2^{3n/2})$	U
			$\mathcal{O}(2^{6n/7})$	$\tilde{\mathcal{O}}(2^{6n/7})$	U
`PMAC+` [Yas11]	$\mathcal{O}(\frac{q^3\ell^3}{2^{2n}})$	$\Omega(2^{2n/3})$	$\mathcal{O}(2^{3n/4})$	$\tilde{\mathcal{O}}(2^{3n/2})$	E
[KLL20]	$\mathcal{O}(\frac{\ell}{2^{n/2}} + \frac{q^{4/3}\ell^{2/3}+q\ell^2}{2^n})$	$\Omega(2^{3n/4})$			
`LightMAC+` [Nai17]	$\mathcal{O}(\frac{q^3}{2^{2n}})$	$\Omega(2^{2n/3})$	$\mathcal{O}(2^{3n/4})$	$\tilde{\mathcal{O}}(2^{3n/2})$	E
[Nai18]	$\mathcal{O}(q_t^2 q_v/2^{2n})$*				
[KLL20]	$\mathcal{O}(\frac{q^{4/3}}{2^n})$	$\Omega(2^{3n/4})$			
`1kPMAC+` [DDN+17]	$\mathcal{O}(\frac{\sigma}{2^n} + \frac{q\sigma^2}{2^{2n}})$	$\Omega(2^{2n/3})$	$\mathcal{O}(2^{3n/4})$	$\tilde{\mathcal{O}}(2^{3n/2})$	E
`3kf9` [ZWS+12]	$\mathcal{O}(\frac{q^3\ell^3}{2^{2n}} + \frac{q\ell}{2^n})$	$\Omega(2^{2n/3})$	$\mathcal{O}(\sqrt[4]{n} \cdot 2^{3n/4})$	$\tilde{\mathcal{O}}(2^{5n/4})$	U
[KLL20]	$\mathcal{O}(\frac{q^{4/3}\ell^{4/3}}{2^n} + \frac{q^2\ell^2}{2^{2n}} + \frac{q^4\ell^6}{2^{3n}})$	$\Omega(2^{3n/4})$			
`1kf9` [DDN+15]	$\mathcal{O}(\frac{q\ell^2}{2^n} + \frac{q^3\ell^4}{2^{2n}} + \frac{q^4\ell^4}{2^{3n}} + \frac{q^4\ell^6}{2^{4n}})$*	$\Omega(2^{2n/3})$*	$\mathcal{O}(2^{n/2})$	$\tilde{\mathcal{O}}(2^{n/2})$	U

6.3 Generalization to double-block hash-then-sum MACs

In the full paper [LNS18] (Appendix J), we generalize this attack to a framework for attacks against double-block hash-then-sum MACs. Table 6.1 shows a summary of our results and how they compare with their respective provable security claims. In particular, we have forgeries attacks with $\mathcal{O}(2^{3n/4})$ short MAC queries against `SUM-ECBC`, `GCM-SIV2`, `PMAC+`, `LightMAC+`, `1kPMAC+`, and `3kf9`. As far as we know, these are the first attacks with less than 2^n blocks of data against these constructions.

These results were an important step towards understanding the actual security of these modes, by providing a security upper bound of $2^{3n/4}$ short queries when the lower bound from the proof was $2^{2n/3}$ queries. Later work has given security proof with the same bound $2^{3n/4}$ [KLL20] for most of these constructions, which closes the gap between proofs and attacks.

Our attacks exploit the fact that the two halves are combined with a sum, where one side depends only on Σ, and the other side depends only on Θ. They do not seem applicable to constructions with more intricate finalization functions, such as `LightMAC+2` [Nai17], or the tweakable block-cipher based constructions `PMAC_TBC` [Nai15], `PMACx` [LN17a], `ZMAC` [IMP+17], or `ZMAC+`[LN17b].

Mistakes in proofs. Our attack against `LighMAC+` contradicts a security bound for `LighMAC+` by Naito [Nai18], because we have an attack with $\mathcal{O}(2^{3n/4})$ MAC queries, and a single verification query. We have also found an attack with only $\mathcal{O}(2^{n/2})$ queries and $\tilde{\mathcal{O}}(2^{n/2})$ operations against `1kf9` [DDN+15], a single key variant of `3kf9` with claimed security up to $2^{2n/3}$ queries. `1kf9` had been withdrawn due to issues with its security proof before our result, but no attack was known previously.

7. Generic Attacks Against MACs in the Quantum Setting

This chapter is based on articles [KLL+16a] and [BLN+21], found in Appendix K and L.

Cryptography would be particularly affected by the development of large-scale quantum computers. In particular, currently used asymmetric cryptographic primitives based on number theory (RSA [RSA78], Diffie-Hellman key exchange [DH76], elliptic-curve cryptography [Mil86; Kob87], ...) would be broken in polynomial time due to Shor's algorithm [Sho97]. This has motivated the cryptographic community to search for quantum-safe solutions: the goal of post-quantum cryptography is to prepare cryptographic primitives to resist quantum adversaries, *i.e.* adversaries with access to a quantum computer. Symmetric cryptography is much less impacted by quantum algorithms. The main result is a quadratic speed-up of exhaustive search using Grover's algorithm [Gro96]. Since the attack is still exponential, it can be countered by just properly dimensioning the primitive.

In this chapter we analyze the security of MAC algorithms in a quantum setting. We consider attacks where an adversary can query an oracle implementing a cryptographic primitive in a quantum superposition of different states. This model gives a lot of power to the adversary, but recent results show that it is nonetheless possible to build secure cryptosystems in this model.

We show that several classical attacks based on finding collisions can be dramatically sped up using a quantum procedure called *Simon's algorithm*: finding a collision requires $\Omega(2^{n/2})$ queries in the classical setting, but when collisions happen with some hidden periodicity, they can be found with only $\mathcal{O}(n)$ queries in the quantum model. We obtain a series of attacks against many standardized MACs: `CBC-MAC`, `PMAC`, `GMAC`, `LightMAC`, ... This is quite surprising compared to the situation with encryption modes: standard modes (`CBC`, `CTR`, `OFB`, `CFB`) are secure with a quantum-secure block ciphers [ATT+16].

7.1 Quantum security models

We present our attacks using quantum algorithms as black boxes, so that our results can be understood without specific knowledge of quantum computing. Nonetheless, we give a basic description of the quantum circuit model in order to define quantum security notions.

The basic computation units are qubits, two-level quantum systems whose state is represented by a *superposition* $\alpha|0\rangle + \beta|1\rangle$, with amplitudes α and β, which is a normalized vector in $\mathbb{C}^2$ (of norm $|\alpha^2| + |\beta^2| = 1$). The state of an n-qubit system belongs to $\mathbb{C}^{2^n}$, its 2^n basis vectors (in the computational basis) are the 2^n n-bit strings. A quantum algorithm is a sequence of unitary operators of $\mathbb{C}^{2^n}$, partial measurements, and oracle calls. We say that a function f is queried *in superposition* if the following unitary operator O_f is made available: $|x\rangle|y\rangle \mapsto |x\rangle|y \oplus f(x)\rangle$. Indeed, this operator allows querying f on any quantum state, thus on any *superposition* of inputs x.

Q1 and Q2 setting. To define quantum security notion, we consider the adversary as a quantum algorithm. Following [KLL+16b], we adopt the Q1 / Q2 terminology to classify quantum attacks on symmetric schemes. In the Q1 setting, the adversary is given only *classical* encryption or decryption query access to black-boxes. In the Q2 setting, the adversary is given *quantum* or *superposition* access, in the sense that a black-box E_k becomes a quantum oracle O_{E_k}. Note that these models have alternative names, for example Q2 security has been called "quantum chosen-plaintext attack" (qCPA) or "quantum security" in previous works.

The Q1 setting is more realistic: the adversary receives classical data, and only uses local quantum computation. This is the setting used by attacks against asymmetric schemes based on Shor's algorithm: the adversary receives the (classical) public key, and uses a quantum computer to recover the private key. Key-recovery attacks using Grover's algorithm are also in the Q1 setting: the adversary receives a pair of known plaintext and ciphertext, and uses quantum computation locally to recover the corresponding key.

The Q2 setting is a simple and natural extension of the Q1 setting, giving more power to the adversary. It encompasses any reasonable model of quantum attacks, so that a security proof in this model gives a high level of assurance. Previous work has shown that it is possible to build secure cryptosystems in this model [BZ13a]. Concretely, it assumes that the legitimate user implements the algorithm on a quantum computer, and that the adversary obtains oracle access.

The study of quantum attacks on symmetric schemes in the Q2 setting was sparkled by the seminal work of Kuwakado and Morii [KM10; KM12], who showed that the 3-round Feistel construction and the Even-Mansour cipher became insecure if exposed to superposition queries. More precisely, they use Simon's algorithm to respectively distinguish the construction and recover the key of the cipher in polynomial time, while classical proofs of security exist.

Unforgeability. In this chapter, we consider forgery attacks: the goal of the attacker is to forge a tag for some arbitrary message, without the knowledge of the secret key. In a classical setting, we assume that the adversary has access to a MAC oracle, and she must predict the MAC of a message that she has not queried. In the quantum setting, we must modify this definition because a superposition query can include all messages simultaneously.

We follow the EUF-qCMA security definition of Boneh and Zhandry [BZ13a]. A message authentication code is broken by a quantum existential forgery attack if after q queries to the cryptographic oracle, the adversary can generate at least $q+1$ valid messages with corresponding tags.

For nonce-based MACs, we assume that the nonce is classical, and chosen randomly by the adversary (as in the IND-qCPA notion [BZ13b]). This sidesteps the issue of defining the uniqueness of a nonce in quantum superposition.

7.2 Simon's algorithm

Simon's algorithm [Sim97] was one of the first algorithms to show an exponential speedup in the quantum setting, and inspired Shor's algorithm. It solves the following problem:

> **Simon's problem**: Given a Boolean function $f : \{0,1\}^n \to \{0,1\}^n$ and the promise that there exists $s \in \{0,1\}^n$ such that for any $(x, y) \in \{0,1\}^n$, $[f(x) = f(y)] \Leftrightarrow [x \oplus y \in \{0^n, s\}]$, the goal is to find s.

This problem can be solved classically by searching for collisions. The optimal time to solve it is therefore $\Theta(2^{n/2})$. On the other hand, Simon's algorithm solves this problem

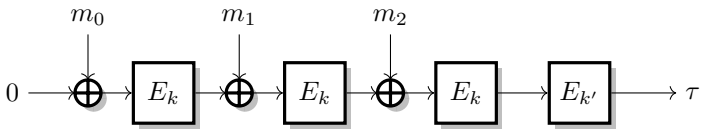


Figure 7.1: Encrypt-last-block `CBC-MAC`.

with quantum complexity $\mathcal{O}(n)$ using superposition queries to f.

The algorithm uses a quantum subroutine that makes a single superposition query to f, and returns a random vector y such that $y \cdot s = 0$. By repeating this subroutine $\mathcal{O}(n)$ times, one obtains $n-1$ independent vectors orthogonal to s with high probability, and s can be recovered using basic linear algebra.

7.2.1 Attack strategy

The general strategy behind our attacks follows the approach used by Kuwakado and Morii. We start with the MAC oracle $F_k : \{0,1\}^n \to \{0,1\}^n$ and exhibit a new function f that satisfies Simon's promise with two additional properties: the adversary should be able to query f in superposition if she has quantum oracle access to F_k, and the knowledge of the string s should be sufficient to break the cryptographic scheme. In most cases, the construction of f corresponds to a classical collision attack.

7.2.2 Dealing with unwanted collisions

In our cryptanalysis scenario, it is not always the case that the promise of Simon's problem is perfectly satisfied. More precisely, by construction, there will always exist an s such that $f(x) = f(x \oplus s)$ for any input x, but there might be additional collisions. If the number of such unwanted collisions is too large, one might not be able to obtain a full rank linear system of equations from Simon's subroutine after $\mathcal{O}(n)$ queries.

In [KLL+16a] (Appendix K), we show that Simon's algorithm runs in time $\mathcal{O}(n)$ under a weaker assumption:

$$\max_{t \notin \{0,s\}} \Pr_x \left[f(x \oplus t) = f(x) \right] \leq \frac{1}{2} \ . \tag{7.1}$$

In practice, we cannot prove this bound, but if it does not hold, then some of the primitives used in the construction are far from ideal. We can show that this happens with low probability, and would imply a classical attack against the system.

7.3 Breaking `CBC-MAC`

`CBC-MAC` is one of the earliest MAC constructions, inspired by the CBC encryption mode [NBS85]. Since the basic `CBC-MAC` is only secure when the queries are prefix-free, there are many variants of `CBC-MAC` to provide security for arbitrary messages. In the following we describe the Encrypt-last-block `CBC-MAC` variant [BKR94], using two keys k and k', but the attack can be easily adapted to other variants. On a message $M = m_0 \,\|\, \ldots \,\|\, m_{L-1}$, `CBC-MAC` is defined as (see Figure 7.1):

$$x_0 = 0 \qquad x_{i+1} = E_k(x_i \oplus m_i) \qquad \texttt{CBC-MAC}(M) = E_{k'}(x_L)$$

`CBC-MAC` is standardized and widely used. It has been proven secure up to the birthday bound [BKR94], assuming that the block cipher is indistinguishable from a random permutation.

Quantum attack. We present a powerful forgery attack on CBC-MAC with very low complexity using superposition queries. We fix two arbitrary message blocks α_0, α_1, with $\alpha_0 \neq \alpha_1$, and we define the following function:

$$\begin{aligned} f : \{0,1\} \times \{0,1\}^n &\to \{0,1\}^n \\ b, x \quad &\mapsto \texttt{CBC-MAC}(\alpha_b \,\|\, x) = E_{k'}\left(E_k\big(x \oplus E_k(\alpha_b)\big)\right). \end{aligned}$$

The function f can be computed with a single call to the cryptographic oracle, and we can build a quantum circuit for f given a black box quantum circuit for $\texttt{CBC-MAC}_k$. Moreover, f satisfies the promise of Simon's problem with $s = 1 \,\|\, E_k(\alpha_0) \oplus E_k(\alpha_1)$:

$$\begin{aligned} f(0,x) &= E_{k'}(E_k(x \oplus E_k(\alpha_1))), \\ f(1,x) &= E_{k'}(E_k(x \oplus E_k(\alpha_0))), \\ f(b,x) &= f(b \oplus 1, x \oplus E_k(\alpha_0) \oplus E_k(\alpha_1)). \end{aligned}$$

More precisely:

$$\begin{aligned} f(b',x') = f(b,x) &\Leftrightarrow x \oplus E_k(\alpha_b) = x' \oplus E_k(\alpha_{b'}) \\ &\Leftrightarrow \begin{cases} x' \oplus x = 0 & \text{if } b' = b \\ x' \oplus x = E_k(\alpha_0) \oplus E_k(\alpha_1) & \text{if } b' \neq b \end{cases} \end{aligned}$$

Therefore, an application of Simon's algorithm returns $E_k(\alpha_0) \oplus E_k(\alpha_1)$. This allows to forge messages easily:

1. Query the tag of $\alpha_0 \,\|\, m_1$ for an arbitrary block m_1;
2. The same tag is valid for $\alpha_1 \,\|\, m_1 \oplus E_k(\alpha_0) \oplus E_k(\alpha_1)$.

In order to break the formal notion of EUF-qCMA security, we must produce $q+1$ valid tags with only q queries to the oracle. Let $q' = \mathcal{O}(n)$ denote the number of quantum queries made to learn $E_k(\alpha_0) \oplus E_k(\alpha_1)$. The attacker just repeats the forgery step $q'+1$ times, in order to produce $2(q'+1)$ messages with valid tags, after a total of $2q'+1$ classical and quantum queries to the cryptographic oracle. Therefore, CBC-MAC is broken by a quantum existential forgery attack.

7.4 Breaking PMAC

PMAC is a parallelizable block-cipher based MAC designed by Rogaway [Rog04]. PMAC is based on the XE construction: the construction uses secret offsets Δ_i and Δ^* derived from the secret key to turn the block cipher into a tweakable block cipher. More precisely, the PMAC algorithm is defined on a message $M = m_0 \,\|\, \ldots \,\|\, m_{L-1}$ as (see Figure 7.2):

$$c_i = E_k(m_i \oplus \Delta_i) \qquad \texttt{PMAC}(M) = E_k\Big(\Delta^* \oplus m_{L-1} \oplus \bigoplus_{i=0}^{L-2} c_i\Big)$$

Some authenticated encryption schemes use variants of PMAC; we denote as PMAC' the variant used in [HKR17] (see Figure 7.3):

$$c_i = E_k(m_i \oplus \Delta_i) \qquad \texttt{PMAC'}(M) = E_k\Big(\Delta^* \oplus \bigoplus_{i=0}^{L-1} c_i\Big)$$

We omit the generation of the secret offsets because it is irrelevant to our attacks.

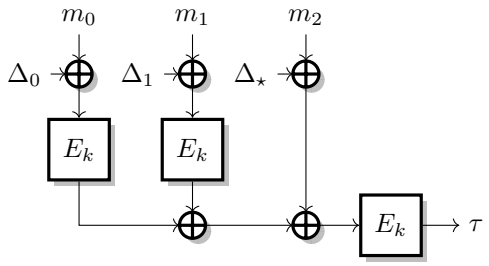


Figure 7.2: PMAC.

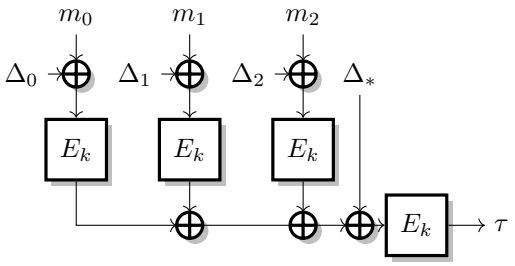


Figure 7.3: PMAC' variant.

Using the CBC-MAC attack. When PMAC is used with two-block messages, it has the same structure as CBC-MAC: $\texttt{PMAC}(m_0 \,\|\, m_1) = E_k^*(m_1 \oplus E_k^0(m_0))$, with $E_k^0(x) = E_k(x \oplus \Delta_0)$ and $E_k^*(x) = E_k(x \oplus \Delta_*)$. Therefore, we can use the attack of the previous section to recover $E_k^0(\alpha_0) \oplus E_k^0(\alpha_1)$ for arbitrary values of α_0 and α_1.

Quantum attack targeting offsets. The CBC-MAC attack is not applicable to the PMAC' variant. We propose an alternative attack to recover the difference between two offsets Δ_i. We use the following function:

$$\begin{aligned} f : \{0,1\}^n &\to \{0,1\}^n \\ x &\mapsto \texttt{PMAC'}(x \,\|\, x) = E_k^*\left(E_k(x \oplus \Delta_0) \oplus E_k(x \oplus \Delta_1)\right). \end{aligned}$$

In particular, it satisfies $f(x \oplus s) = f(x)$ with $s = \Delta_0 \oplus \Delta_1$. This function does not satisfy the strict promise of Simon's algorithm because it has additional collisions, but we can show that the condition (7.1) is satisfied if E is a good block cipher. Therefore, we apply Simon's algorithm to recover $\Delta_0 \oplus \Delta_1$. We obtain forgeries as follows:

1. Query the tag of $m \,\|\, m$ for an arbitrary block m;
2. The same tag is valid for $m \oplus \Delta_0 \oplus \Delta_1 \,\|\, m \oplus \Delta_0 \oplus \Delta_1$.

7.5 Breaking GMAC

GMAC is the underlying MAC of the widely used GCM standard, designed by McGrew and Viega [MV04], and standardized by NIST. GMAC follows the Wegman-Carter construction [CW77]: it is built from a universal hash function, using polynomial evaluation in a Galois field. As opposed to the constructions of the previous sections, GMAC is a randomized MAC; it requires a second input N, which must be non-repeating (a nonce). GMAC is essentially defined as:

$$\begin{aligned} \texttt{GMAC}(N, M) &= \texttt{GHASH}(M \,\|\, \mathsf{len}(M)) \oplus E_k(N||1) \\ \texttt{GHASH}(M) &= \bigoplus_{i=0}^{\mathsf{len}(M)-1} m_i \odot H^{\mathsf{len}(M)-i} \quad \text{with } H = E_k(0), \end{aligned}$$

where $\mathsf{len}(M)$ is the length of M, and $\odot$ is the finite-field multiplication.

Quantum attack. When the polynomial is evaluated with Horner's rule, the structure of GMAC is similar to that of CBC-MAC (see Figure 7.4). For a two-block message, we have $\texttt{GMAC}(m_0 \,\|\, m_1) = \big((m_0 \odot H) \oplus m_1\big) \odot H \oplus E_k(N \,\|\, 1)$. Therefore, we use the same function as in the CBC-MAC attack, with fixed blocks α_0 and α_1:

$$\begin{aligned} f_N : \{0,1\} \times \{0,1\}^n &\to \{0,1\}^n \\ b, x &\mapsto \texttt{GMAC}(N, \alpha_b \,\|\, x) = \alpha_b \odot H^2 \oplus x \odot H \oplus E_k(N||1). \end{aligned}$$

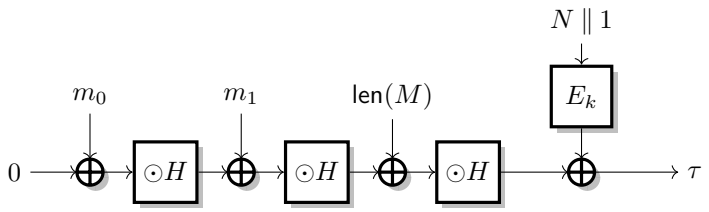


Figure 7.4: GMAC.

In particular, we have:

$$f_N(b', x') = f(b, x) \Leftrightarrow \alpha_b \odot H^2 \oplus x \odot H = \alpha_{b'} \odot H^2 \oplus x' \odot H$$
$$\Leftrightarrow \begin{cases} x' \oplus x = 0 & \text{if } b' = b \\ x' \oplus x = (\alpha_0 \oplus \alpha_1) \odot H & \text{if } b' \neq b \end{cases}$$

Therefore f_N satisfies the promise of Simon's algorithm with $s = 1 \parallel (\alpha_0 \oplus \alpha_1) \odot H$.

Dealing with the nonce. Since N is a nonce we can only query f_N once.[1] Therefore, we cannot apply Simon's algorithm as is, because this requires $\mathcal{O}(n)$ queries to the *same* function. However, a single step of Simon's algorithm requires a single query to the f_N function, and returns a vector orthogonal to s, for any random choice of N (the secret period s is independent of N). Therefore, we recover $(\alpha_0 \oplus \alpha_1) \odot H$ after $\mathcal{O}(n)$ steps, even if each step uses a different value of N. Then, we deduce H easily, and generate forgeries using H:

1. Query the tag of $N, m_0 \parallel m_1$ for arbitrary blocks m_0, m_1 (under a random nonce N).
2. The same tag is valid for $m_0 \oplus 1 \parallel m_1 \oplus H$ (with the same nonce N).

As for CBC-MAC, repeating these two steps leads to an existential forgery attack.

7.6 Breaking ΘCB3

ΘCB3 [KR11] is the tweakable block-cipher variant of the authenticated encryption scheme OCB3; OCB3 has been standardized by IETF [KR14]. ΘCB3 has the advantage of being secure up to 2^n queries (beyond the birthday bound). When used with an empty message, it becomes a nonce-based MAC:

$$c_i = E_k^i(m_i) \qquad \text{ΘCB3-MAC}(N, M) = F_k(N) \oplus \bigoplus_{i=0}^{L-1} c_i$$

In this description, E is a tweakable block-cipher: each E^i is assumed to be an independent block cipher. We omit the details of the F_k function for simplicity.

Quantum linearization attack. All the attacks described so far exploit the XOR between a secret value (the internal state or a secret offset) and a value chosen by the attacker. Variants of those attacks can be mounted against many MACs with this property. However, in ΘCB3 there is no XOR between the message and a sensitive value, and we need a different type of attack.

Our key idea is to *linearize* the MAC by restricting the block inputs so that the internal state $\Sigma = \bigoplus_{i=0}^{L-1} E_k^i(m_i)$ is an affine function. We fix two arbitrary message blocks α_0, α_1,

[1] Indeed, we use a quantum security definition where N is a random classical value chosen by the oracle.

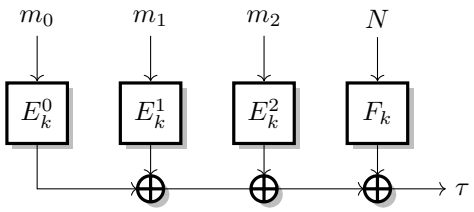


Figure 7.5: `ΘCB3` used as a MAC.

with $\alpha_0 \neq \alpha_1$, and we consider messages of the form $M_b = \alpha_{b[0]} \,\|\, \alpha_{b[1]} \,\|\, \alpha_{b[2]} \,\|\, \ldots \,\|\, \alpha_{b[L-1]}$ with $b \in \{0,1\}^L$. We observe that the internal state is an affine function of b:

$$\Sigma(M_b) = \bigoplus_{i=0}^{L} E_k^i(\alpha_{b[i]}) = \bigoplus_{i=0}^{L} \left(E_k^i(\alpha_0) \oplus b[i] \odot (E_k^i(\alpha_0) \oplus E_k^i(\alpha_1))\right)$$

In particular, with $L > n$ the function $b \mapsto \Sigma(M_b)$ necessarily has a non-trivial kernel: there exist an L-bit vector $s \neq 0$ with $\Sigma(M_{b\oplus s}) = \Sigma(M_b)$ for all b. With $L = n + 1$, s is likely to be unique.

Therefore, we define the following function:

$$\begin{aligned} f_N : \{0,1\}^L &\to \{0,1\}^n \\ b &\mapsto \texttt{ΘCB3-MAC}(N, M_b) = F_k(N) \oplus \Sigma(M_b) \end{aligned}$$

The function has period s, so that we recover it using Simon's algorithm. We obtain forgeries as follows:

1. Query the tag of $N, M_0 \,\|\, m$ for an arbitrary block m (under a random nonce N).
2. The same tag is valid for $M_s \,\|\, m$ (with the same nonce N).

Alternative quantum attacks. We observe that the function f_N in `ΘCB3` is actually an affine function. Therefore, we can mount alternative attacks using other quantum algorithms: Deutsch's algorithm [Deu85] or Bernstein-Vazirani's algorithm [BV97]. This is the first application of those algorithms in the context of symmetric cryptography.

7.7 Conclusion

These techniques apply to most of the block-cipher-based MACs proposed in the literature: we obtain polynomial attacks in the Q2 setting against `CBC-MAC`, `PMAC`, `PMAC+`, `GMAC`, `LightMAC`, `LightMAC+`, `PolyMAC`, `ZMAC`, `PMAC_TBC3k`. Similarly, we can break the authentication of the authenticated encryption schemes `OCB`, `ΘCB3`, `GCM`, `GCM-SIV`, `GCM-SIV2`, `EAX`. We also found an efficient attack against `Poly1305` using Shor's algorithm based on similar ideas.

Those attacks are not a practical threat because they use the Q2 model, assuming that the adversary can query a MAC oracle with a quantum superposition of messages. However, this is a surprising result, because all those constructions have classical proof of security and we didn't expect an exponential speedup in the quantum model for symmetric cryptosystems. In particular, this partially answers an open question by Boneh and Zhandry in 2013 [BZ13b]: "Do the CBC-MAC or NMAC constructions give quantum-secure PRFs?".

Quantum-secure MACs. Those attacks also motivate the search for efficient quantum-secure MACs. In [SY17], Song and Yun have proven that HMAC and NMAC are secure against an exponential number of quantum queries. Therefore, when we designed

Saturnin [CDL+20], an authenticated encryption scheme aiming for quantum security, we used a mode based on HMAC.

Building a *parallelizable* quantum-secure MAC is more challenging. Indeed, the linearization technique is applicable to a large class of constructions, as long as they can be written in form $G_N(\bigoplus F_i(m_i))$ with F_i arbitrary functions and G_N a function potentially depending on a nonce. This includes most of the proposed parallelizable MACs. In [BBC+21] we propose a new authenticated encryption mode that is efficient, parallelizable, and quantum secure: QCB. QCB is similar to ΘCB3, but we make all tweakable block-cipher calls dependent on the nonce to avoid this class of attacks.

Bibliography

[ABD+15] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann. "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice". In: *ACM CCS 2015*. Ed. by I. Ray, N. Li, and C. Kruegel. ACM Press, Oct. 2015, pp. 5–17. DOI: 10.1145/2810103.2813707.

[ABP+13] N. J. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. N. Schuldt. "On the Security of RC4 in TLS". In: *USENIX Security 2013*. Ed. by S. T. King. USENIX Association, Aug. 2013, pp. 305–320. URL: https://www.usenix.org/system/files/conference/usenixsecurity13/sec13-paper_alfardan.pdf.

[AES] National Institute for Standards and Technology. *Advanced Encryption Standard (AES)*. FIPS 197. Nov. 2001. DOI: https://doi.org/10.6028/NIST.FIPS.197-upd1.

[AHM+10] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan. *SHA-3 proposal BLAKE*. Submission to NIST. 2008/2010. URL: http://131002.net/blake/blake.pdf.

[Ajt96] M. Ajtai. "Generating Hard Instances of Lattice Problems (Extended Abstract)". In: *28th ACM STOC*. ACM Press, May 1996, pp. 99–108. DOI: 10.1145/237814.237838.

[Alb17] A. Albertini. *Exploiting Hash collisions*. Presented at Black Alps 2017. Nov. 2017. URL: https://github.com/corkami/collisions.

[AP13a] N. AlFardan and K. Paterson. *Lucky Thirteen: Breaking the TLS and DTLS Record Protocols*. Blog post. Archived from the original. Feb. 2013. URL: https://web.archive.org/web/20230327103922/http://www.isg.rhul.ac.uk/tls/lucky13.html.

[AP13b] N. J. AlFardan and K. G. Paterson. "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols". In: *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2013, pp. 526–540. DOI: 10.1109/SP.2013.42.

[ATT+16] M. V. Anand, E. E. Targhi, G. N. Tabia, and D. Unruh. "Post-Quantum Security of the CBC, CFB, OFB, CTR, and XTS Modes of Operation". In: *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016*. Ed. by T. Takagi. Springer, Heidelberg, 2016, pp. 44–63. DOI: 10.1007/978-3-319-29360-8_4.

[BBB+20] D. Bellizia, F. Berti, O. Bronchain, G. Cassiers, S. Duval, C. Guo, G. Leander, G. Leurent, I. Levi, C. Momin, O. Pereira, T. Peters, F.-X. Standaert, B. Udvarhelyi, and F. Wiemer. "Spook: Sponge-Based Leakage-Resistant Authenticated Encryption with a Masked Tweakable Block Cipher". In: *IACR Trans. Symm. Cryptol.* 2020.S1 (2020), pp. 295–349. ISSN: 2519-173X. DOI: 10.13154/tosc.v2020.iS1.295-349.

[BBC+21] R. Bhaumik, X. Bonnetain, A. Chailloux, G. Leurent, M. Naya-Plasencia, A. Schrottenloher, and Y. Seurin. "QCB: Efficient Quantum-Secure Authenticated Encryption". In: *ASIACRYPT 2021, Part I*. Ed. by M. Tibouchi and H. Wang. Vol. 13090. LNCS. Springer, Heidelberg, Dec. 2021, pp. 668–698. DOI: 10.1007/978-3-030-92062-3_23.

[BBD+15] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. "A Messy State of the Union: Taming the Composite State Machines of TLS". In: *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2015, pp. 535–552. DOI: 10.1109/SP.2015.39.

[BBF+16] K. Bhargavan, C. Brzuska, C. Fournet, M. Green, M. Kohlweiss, and S. Zanella-Béguelin. "Downgrade Resilience in Key-Exchange Protocols". In: *2016 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2016, pp. 506–525. DOI: 10.1109/SP.2016.37.

[BBK08] E. Barkan, E. Biham, and N. Keller. "Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication". In: *Journal of Cryptology* 21.3 (July 2008), pp. 392–429. DOI: 10.1007/s00145-007-9001-y.

[BBL+15] A. Banerjee, H. Brenner, G. Leurent, C. Peikert, and A. Rosen. "SPRING: Fast Pseudorandom Functions from Rounded Ring Products". In: *FSE 2014*. Ed. by C. Cid and C. Rechberger. Vol. 8540. LNCS. Springer, Heidelberg, Mar. 2015, pp. 38–57. DOI: 10.1007/978-3-662-46706-0_3.

[BBL+22] A. Bariant, C. Bouvier, G. Leurent, and L. Perrin. "Algebraic Attacks against Some Arithmetization-Oriented Primitives". In: *IACR Trans. Symm. Cryptol.* 2022.3 (2022), pp. 73–101. DOI: 10.46586/tosc.v2022.i3.73-101.

[BC04] E. Biham and R. Chen. "Near-Collisions of SHA-0". In: *CRYPTO 2004*. Ed. by M. Franklin. Vol. 3152. LNCS. Springer, Heidelberg, Aug. 2004, pp. 290–305. DOI: 10.1007/978-3-540-28628-8_18.

[BCD+20] T. Beyne, A. Canteaut, I. Dinur, M. Eichlseder, G. Leander, G. Leurent, M. Naya-Plasencia, L. Perrin, Y. Sasaki, Y. Todo, and F. Wiemer. "Out of Oddity - New Cryptanalytic Techniques Against Symmetric Primitives Optimized for Integrity Proof Systems". In: *CRYPTO 2020, Part III*. Ed. by D. Micciancio and T. Ristenpart. Vol. 12172. LNCS. Springer, Heidelberg, Aug. 2020, pp. 299–328. DOI: 10.1007/978-3-030-56877-1_11.

[BCJ+05] E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet, and W. Jalby. "Collisions of SHA-0 and Reduced SHA-1". In: *EUROCRYPT 2005*. Ed. by R. Cramer. Vol. 3494. LNCS. Springer, Heidelberg, May 2005, pp. 36–57. DOI: 10.1007/11426639_3.

[BCK96] M. Bellare, R. Canetti, and H. Krawczyk. "Keying Hash Functions for Message Authentication". In: *CRYPTO'96*. Ed. by N. Koblitz. Vol. 1109. LNCS. Springer, Heidelberg, Aug. 1996, pp. 1–15. DOI: 10.1007/3-540-68697-5_1.

[BD07] E. Biham and O. Dunkelman. *A Framework for Iterative Hash Functions - HAIFA*. Cryptology ePrint Archive, Report 2007/278. https://eprint.iacr.org/2007/278. 2007.

[BDG+20] Z. Bao, I. Dinur, J. Guo, G. Leurent, and L. Wang. "Generic Attacks on Hash Combiners". In: *Journal of Cryptology* 33.3 (July 2020), pp. 742–823. DOI: 10.1007/s00145-019-09328-w.

[BDJ+97] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. "A Concrete Security Treatment of Symmetric Encryption". In: *38th FOCS*. IEEE Computer Society Press, Oct. 1997, pp. 394–403. DOI: 10.1109/SFCS.1997.646128.

[BDL+21] C. Beierle, P. Derbez, G. Leander, G. Leurent, H. Raddum, Y. Rotella, D. Rupprecht, and L. Stennes. "Cryptanalysis of the GPRS Encryption Algorithms GEA-1 and GEA-2". In: *EUROCRYPT 2021, Part II*. Ed. by A. Canteaut and F.-X. Standaert. Vol. 12697. LNCS. Springer, Heidelberg, Oct. 2021, pp. 155–183. DOI: 10.1007/978-3-030-77886-6_6.

[Ber05] D. J. Bernstein. "The Poly1305-AES Message-Authentication Code". In: *FSE 2005*. Ed. by H. Gilbert and H. Handschuh. Vol. 3557. LNCS. Springer, Heidelberg, Feb. 2005, pp. 32–49. DOI: 10.1007/11502760_3.

[Ber08] D. J. Bernstein. "The Salsa20 Family of Stream Ciphers". In: *New Stream Cipher Designs: The eSTREAM Finalists*. Ed. by M. Robshaw and O. Billet. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 84–97. ISBN: 978-3-540-68351-3. DOI: 10.1007/978-3-540-68351-3_8.

[BKR94] M. Bellare, J. Kilian, and P. Rogaway. "The Security of Cipher Block Chaining". In: *CRYPTO'94*. Ed. by Y. Desmedt. Vol. 839. LNCS. Springer, Heidelberg, Aug. 1994, pp. 341–358. DOI: 10.1007/3-540-48658-5_32.

[BL16a] K. Bhargavan and G. Leurent. "On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN". In: *ACM CCS 2016*. Ed. by E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi. ACM Press, Oct. 2016, pp. 456–467. DOI: 10.1145/2976749.2978423.

[BL16b] K. Bhargavan and G. Leurent. "Transcript Collision Attacks: Breaking Authentication in TLS, IKE and SSH". In: *NDSS 2016*. The Internet Society, Feb. 2016. DOI: 10.14722/ndss.2016.23418.

[BL23] A. Bariant and G. Leurent. "Truncated Boomerang Attacks and Application to AES-Based Ciphers". In: *EUROCRYPT 2023, Part IV*. Ed. by C. Hazay and M. Stam. Vol. 14007. LNCS. Springer, Heidelberg, Apr. 2023, pp. 3–35. DOI: 10.1007/978-3-031-30634-1_1.

[BLN+21] X. Bonnetain, G. Leurent, M. Naya-Plasencia, and A. Schrottenloher. "Quantum Linearization Attacks". In: *ASIACRYPT 2021, Part I*. Ed. by M. Tibouchi and H. Wang. Vol. 13090. LNCS. Springer, Heidelberg, Dec. 2021, pp. 422–452. DOI: 10.1007/978-3-030-92062-3_15.

[BLR13] A. Biryukov, G. Leurent, and A. Roy. "Cryptanalysis of the "Kindle" Cipher". In: *SAC 2012*. Ed. by L. R. Knudsen and H. Wu. Vol. 7707. LNCS. Springer, Heidelberg, Aug. 2013, pp. 86–103. DOI: 10.1007/978-3-642-35999-6_7.

[BS91] E. Biham and A. Shamir. "Differential Cryptanalysis of DES-like Cryptosystems". In: *Journal of Cryptology* 4.1 (Jan. 1991), pp. 3–72. DOI: 10.1007/BF00630563.

[BV97] E. Bernstein and U. V. Vazirani. "Quantum Complexity Theory". In: *SIAM J. Comput.* 26.5 (1997), pp. 1411–1473. DOI: 10.1137/S0097539796300921. URL: https://doi.org/10.1137/S0097539796300921.

[BWG+17] Z. Bao, L. Wang, J. Guo, and D. Gu. "Functional Graph Revisited: Updates on (Second) Preimage Attacks on Hash Combiners". In: *CRYPTO 2017, Part II*. Ed. by J. Katz and H. Shacham. Vol. 10402. LNCS. Springer, Heidelberg, Aug. 2017, pp. 404–427. DOI: 10.1007/978-3-319-63715-0_14.

[BZ13a] D. Boneh and M. Zhandry. "Quantum-Secure Message Authentication Codes". In: *EUROCRYPT 2013*. Ed. by T. Johansson and P. Q. Nguyen. Vol. 7881. LNCS. Springer, Heidelberg, May 2013, pp. 592–608. DOI: 10.1007/978-3-642-38348-9_35.

[BZ13b] D. Boneh and M. Zhandry. "Secure Signatures and Chosen Ciphertext Security in a Quantum Computing World". In: *CRYPTO 2013, Part II*. Ed. by R. Canetti and J. A. Garay. Vol. 8043. LNCS. Springer, Heidelberg, Aug. 2013, pp. 361–379. DOI: 10.1007/978-3-642-40084-1_21.

[CDL+20] A. Canteaut, S. Duval, G. Leurent, M. Naya-Plasencia, L. Perrin, T. Pornin, and A. Schrottenloher. "Saturnin: a suite of lightweight symmetric algorithms for post-quantum security". In: *IACR Trans. Symm. Cryptol.* 2020.S1 (2020), pp. 160–207. ISSN: 2519-173X. DOI: 10.13154/tosc.v2020.iS1.160-207.

[CDL16] A. Canteaut, S. Duval, and G. Leurent. "Construction of Lightweight S-Boxes Using Feistel and MISTY Structures". In: *SAC 2015*. Ed. by O. Dunkelman and L. Keliher. Vol. 9566. LNCS. Springer, Heidelberg, Aug. 2016, pp. 373–393. DOI: 10.1007/978-3-319-31301-6_22.

[CDM+05] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. "Merkle-Damgård Revisited: How to Construct a Hash Function". In: *CRYPTO 2005*. Ed. by V. Shoup. Vol. 3621. LNCS. Springer, Heidelberg, Aug. 2005, pp. 430–448. DOI: 10.1007/11535218_26.

[Cen23] Censys. *Host Search*. Archived version at https://web.archive.org/web/20231108185913/https://search.censys.io/search/report?resource=hosts&q=services.extended_service_name%3A+HTTPS&virtual_hosts=EXCLUDE&field=services.tls.cipher_selected&num_buckets=50. Accessed on 2023-11-08. Nov. 2023. URL: https://search.censys.io/search/report?resource=hosts&q=services.extended_service_name%3A+HTTPS&virtual_hosts=EXCLUDE&field=services.tls.cipher_selected&num_buckets=50.

[CJ98] F. Chabaud and A. Joux. "Differential Collisions in SHA-0". In: *CRYPTO'98*. Ed. by H. Krawczyk. Vol. 1462. LNCS. Springer, Heidelberg, Aug. 1998, pp. 56–71. DOI: 10.1007/BFb0055720.

[CW77] L. Carter and M. N. Wegman. "Universal Classes of Hash Functions (Extended Abstract)". In: *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*. Ed. by J. E. Hopcroft, E. P. Friedman, and M. A. Harrison. ACM, 1977, pp. 106–112. DOI: 10.1145/800105.803400.

[DAM+15] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman. "A Search Engine Backed by Internet-Wide Scanning". In: *ACM CCS 2015*. Ed. by I. Ray, N. Li, and C. Kruegel. ACM Press, Oct. 2015, pp. 542–553. DOI: 10.1145/2810103.2813703.

[Dam90] I. Damgård. "A Design Principle for Hash Functions". In: *CRYPTO'89*. Ed. by G. Brassard. Vol. 435. LNCS. Springer, Heidelberg, Aug. 1990, pp. 416–427. DOI: 10.1007/0-387-34805-0_39.

[dB94] B. den Boer and A. Bosselaers. "Collisions for the Compressin Function of MD5". In: *EUROCRYPT'93*. Ed. by T. Helleseth. Vol. 765. LNCS. Springer, Heidelberg, May 1994, pp. 293–304. DOI: 10.1007/3-540-48285-7_26.

[DD13] V. Dolmatov and A. Degtyarev. *GOST R 34.11-2012: Hash Function*. RFC 6986. Aug. 2013. DOI: 10.17487/RFC6986.

[DDN+15] N. Datta, A. Dutta, M. Nandi, G. Paul, and L. Zhang. *Building Single-Key Beyond Birthday Bound Message Authentication Code*. Cryptology ePrint Archive, Report 2015/958. https://eprint.iacr.org/2015/958. 2015.

[DDN+17] N. Datta, A. Dutta, M. Nandi, G. Paul, and L. Zhang. "Single Key Variant of PMAC_Plus". In: *IACR Trans. Symm. Cryptol.* 2017.4 (2017), pp. 268–305. ISSN: 2519-173X. DOI: 10.13154/tosc.v2017.i4.268-305.

[DDN+18] N. Datta, A. Dutta, M. Nandi, and G. Paul. "Double-block Hash-then-Sum: A Paradigm for Constructing BBB Secure PRF". In: *IACR Trans. Symm. Cryptol.* 2018.3 (2018), pp. 36–92. ISSN: 2519-173X. DOI: 10.13154/tosc.v2018.i3.36-92.

[Deu85] D. Deutsch. "Quantum theory, the Church–Turing principle and the universal quantum computer". In: *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 400.1818 (1985), pp. 97–117. DOI: 10.1098/rspa.1985.0070.

[DH76] W. Diffie and M. E. Hellman. "New Directions in Cryptography". In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638.

[DH79] W. Diffie and M. E. Hellman. "Privacy and authentication: An introduction to cryptography". In: *Proceedings of the IEEE* 67.3 (1979), pp. 397–427. DOI: 10.1109/PROC.1979.11256.

[DL05] M. Daum and S. Lucks. *Hash Collisions (The Poisoned Message Attack) "The Story of Alice and her Boss"*. Presented at the rump session of Eurocrypt 2005. 2005. URL: https://www.sos.cs.ru.nl/applications/courses/security2015/md5collisions/index.html.

[DL14] I. Dinur and G. Leurent. "Improved Generic Attacks against Hash-Based MACs and HAIFA". In: *CRYPTO 2014, Part I*. Ed. by J. A. Garay and R. Gennaro. Vol. 8616. LNCS. Springer, Heidelberg, Aug. 2014, pp. 149–168. DOI: 10.1007/978-3-662-44371-2_9.

[DL16] I. Dinur and G. Leurent. "Improved Generic Attacks Against Hash-Based MACs and HAIFA". In: *Algorithmica* (2016), pp. 1–35. ISSN: 1432-0541. DOI: 10.1007/s00453-016-0236-6.

[DL18] S. Duval and G. Leurent. "MDS Matrices with Lightweight Circuits". In: *IACR Trans. Symm. Cryptol.* 2018.2 (2018), pp. 48–78. ISSN: 2519-173X. DOI: 10.13154/tosc.v2018.i2.48-78.

[Dob96] H. Dobbertin. "Cryptanalysis of MD4". In: *FSE'96*. Ed. by D. Gollmann. Vol. 1039. LNCS. Springer, Heidelberg, Feb. 1996, pp. 53–69. DOI: 10.1007/3-540-60865-6_43.

[Dob97] H. Dobbertin. "RIPEMD with Two-Round Compress Function is Not Collision-Free". In: *Journal of Cryptology* 10.1 (Dec. 1997), pp. 51–70. DOI: 10.1007/s001459900019.

[Dol10] V. Dolmatov. *GOST R 34.11-94: Hash Function Algorithm*. RFC 5831. Mar. 2010. DOI: 10.17487/RFC5831.

[DR06] C. De Cannière and C. Rechberger. "Finding SHA-1 Characteristics: General Results and Applications". In: *ASIACRYPT 2006*. Ed. by X. Lai and K. Chen. Vol. 4284. LNCS. Springer, Heidelberg, Dec. 2006, pp. 1–20. DOI: 10.1007/11935230_1.

[DR11] T. Duong and J. Rizzo. *Here come the ⊕ ninjas*. May 2011. URL: https://bugzilla.mozilla.org/attachment.cgi?id=540839.

[EKM+08] T. Eisenbarth, T. Kasper, A. Moradi, C. Paar, M. Salmasizadeh, and M. T. M. Shalmani. "On the Power of Power Analysis in the Real World: A Complete Break of the KeeLoqCode Hopping Scheme". In: *CRYPTO 2008*. Ed. by D. Wagner. Vol. 5157. LNCS. Springer, Heidelberg, Aug. 2008, pp. 203–220. DOI: 10.1007/978-3-540-85174-5_12.

[FLN+20] A. Flórez-Gutiérrez, G. Leurent, M. Naya-Plasencia, L. Perrin, A. Schrottenloher, and F. Sibleyras. "New Results on Gimli: Full-Permutation Distinguishers and Improved Collisions". In: *ASIACRYPT 2020, Part I*. Ed. by S. Moriai and H. Wang. Vol. 12491. LNCS. Springer, Heidelberg, Dec. 2020, pp. 33–63. DOI: 10.1007/978-3-030-64837-4_2.

[FLN+21] A. Flórez-Gutiérrez, G. Leurent, M. Naya-Plasencia, L. Perrin, A. Schrottenloher, and F. Sibleyras. "Internal Symmetries and Linear Properties: Full-permutation Distinguishers and Improved Collisions on Gimli". In: *Journal of Cryptology* 34.4 (Oct. 2021), p. 45. DOI: 10.1007/s00145-021-09413-z.

[FLS+10] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. *The Skein Hash Function Family*. Submission to NIST. 2008/2010. URL: http://skein-hash.info.

[FLS15] T. Fuhr, G. Leurent, and V. Suder. "Collision Attacks Against CAESAR Candidates - Forgery and Key-Recovery Against AEZ and Marble". In: *ASIACRYPT 2015, Part II*. Ed. by T. Iwata and J. H. Cheon. Vol. 9453. LNCS. Springer, Heidelberg, Nov. 2015, pp. 510–532. DOI: 10.1007/978-3-662-48800-3_21.

[FM01] S. R. Fluhrer and D. A. McGrew. "Statistical Analysis of the Alleged RC4 Keystream Generator". In: *FSE 2000*. Ed. by B. Schneier. Vol. 1978. LNCS. Springer, Heidelberg, Apr. 2001, pp. 19–30. DOI: 10.1007/3-540-44706-7_2.

[FMS01] S. R. Fluhrer, I. Mantin, and A. Shamir. "Weaknesses in the Key Scheduling Algorithm of RC4". In: *SAC 2001*. Ed. by S. Vaudenay and A. M. Youssef. Vol. 2259. LNCS. Springer, Heidelberg, Aug. 2001, pp. 1–24. DOI: 10.1007/3-540-45537-X_1.

[FO90] P. Flajolet and A. M. Odlyzko. "Random Mapping Statistics". In: *EUROCRYPT'89*. Ed. by J.-J. Quisquater and J. Vandewalle. Vol. 434. LNCS. Springer, Heidelberg, Apr. 1990, pp. 329–354. DOI: 10.1007/3-540-46885-4_34.

[FS09] P. Flajolet and R. Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009. ISBN: 978-0-521-89806-5. URL: https://ac.cs.princeton.edu/home/.

[FSK11] N. Ferguson, B. Schneier, and T. Kohno. *Cryptography engineering: design principles and practical applications*. John Wiley & Sons, 2011. ISBN: 9780470474242. DOI: 10.1002/9781118722367.

[GHK+23] H. Gilbert, R. Heim Boissier, L. Khati, and Y. Rotella. "Generic Attack on Duplex-Based AEAD Modes Using Random Function Statistics". In: *EUROCRYPT 2023, Part IV*. Ed. by C. Hazay and M. Stam. Vol. 14007. LNCS. Springer, Heidelberg, Apr. 2023, pp. 348–378. DOI: 10.1007/978-3-031-30634-1_12.

[GIS06] M. Gebhardt, G. Illies, and W. Schindler. "A Note on the Practical Value of Single Hash Collisions for Special File Formats". In: *Sicherheit 2006: Sicherheit - Schutz und Zuverlässigkeit, Beiträge der 3. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.v. (GI), 20.-22. Februar 2006 in Magdeburg*. Ed. by J. Dittmann. Vol. P-77. LNI. GI, 2006, pp. 333–344. URL: https://dl.gi.de/handle/20.500.12116/24792.

[GK08] P. Gauravaram and J. Kelsey. "Linear-XOR and Additive Checksums Don't Protect Damgård-Merkle Hashes from Generic Attacks". In: *CT-RSA 2008*. Ed. by T. Malkin. Vol. 4964. LNCS. Springer, Heidelberg, Apr. 2008, pp. 36–51. DOI: 10.1007/978-3-540-79263-5_3.

[GKM+08] F. D. Garcia, G. de Koning Gans, R. Muijrers, P. van Rossum, R. Verdult, R. Wichers Schreur, and B. Jacobs. "Dismantling MIFARE Classic". In: *ESORICS 2008*. Ed. by S. Jajodia and J. López. Vol. 5283. LNCS. Springer, Heidelberg, Oct. 2008, pp. 97–114. DOI: 10.1007/978-3-540-88313-5_7.

[GKS13] F. Giesen, F. Kohlar, and D. Stebila. "On the security of TLS renegotiation". In: *ACM CCS 2013*. Ed. by A.-R. Sadeghi, V. D. Gligor, and M. Yung. ACM Press, Nov. 2013, pp. 387–398. DOI: 10.1145/2508859.2516694.

[GLS+14] V. Grosso, G. Leurent, F.-X. Standaert, K. Varici, F. Durvaux, L. Gaspar, and S. Kerckhof. *SCREAM & iSCREAM: Side-Channel Resistant Authenticated Encryption with Masking*. Submitted to the CAESAR competition. Mar. 2014. URL: http://perso.uclouvain.be/fstandae/SCREAM/.

[GLS+15] V. Grosso, G. Leurent, F.-X. Standaert, and K. Varici. "LS-Designs: Bitslice Encryption for Efficient Masked Software Implementations". In: *FSE 2014*. Ed. by C. Cid and C. Rechberger. Vol. 8540. LNCS. Springer, Heidelberg, Mar. 2015, pp. 18–37. DOI: 10.1007/978-3-662-46706-0_2.

[GM84] S. Goldwasser and S. Micali. "Probabilistic Encryption". In: *Journal of Computer and System Sciences* 28.2 (1984), pp. 270–299. DOI: 10.1016/0022-0000(84)90070-9.

[Gol97] J. D. Golic. "Cryptanalysis of Alleged A5 Stream Cipher". In: *EUROCRYPT'97*. Ed. by W. Fumy. Vol. 1233. LNCS. Springer, Heidelberg, May 1997, pp. 239–255. DOI: 10.1007/3-540-69053-0_17.

[GPS+14] J. Guo, T. Peyrin, Y. Sasaki, and L. Wang. "Updates on Generic Attacks against HMAC and NMAC". In: *CRYPTO 2014, Part I*. Ed. by J. A. Garay and R. Gennaro. Vol. 8616. LNCS. Springer, Heidelberg, Aug. 2014, pp. 131–148. DOI: 10.1007/978-3-662-44371-2_8.

[GPv15] C. Garman, K. G. Paterson, and T. van der Merwe. "Attacks Only Get Better: Password Recovery Attacks Against RC4 in TLS". In: *USENIX Security 2015*. Ed. by J. Jung and T. Holz. USENIX Association, Aug. 2015, pp. 113–128. URL: https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-garman.pdf.

[Gro96] L. K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search". In: *28th ACM STOC*. ACM Press, May 1996, pp. 212–219. DOI: 10.1145/237814.237866.

[HBK+11] R. Holz, L. Braun, N. Kammenhuber, and G. Carle. "The SSL landscape: a thorough analysis of the x. 509 PKI using active and passive measurements". In: *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. 2011, pp. 427–444. DOI: 10.1145/2068816.2068856.

[HKR17] V. T. Hoang, T. Krovetz, and P. Rogaway. *AEZ v5: Authenticated Encryption by Enciphering*. Submission to the CAESAR competition. Mar. 2017. URL: https://competitions.cr.yp.to/round3/aezv5.pdf.

[HP08] H. Handschuh and B. Preneel. "Key-Recovery Attacks on Universal Hash Function Based MAC Algorithms". In: *CRYPTO 2008*. Ed. by D. Wagner. Vol. 5157. LNCS. Springer, Heidelberg, Aug. 2008, pp. 144–161. DOI: 10.1007/978-3-540-85174-5_9.

[HRR+15] V. T. Hoang, R. Reyhanitabar, P. Rogaway, and D. Vizár. "Online Authenticated-Encryption and its Nonce-Reuse Misuse-Resistance". In: *CRYPTO 2015, Part I*. Ed. by R. Gennaro and M. J. B. Robshaw. Vol. 9215. LNCS. Springer, Heidelberg, Aug. 2015, pp. 493–517. DOI: 10.1007/978-3-662-47989-6_24.

[HS08] J. J. Hoch and A. Shamir. "On the Strength of the Concatenated Hash Combiner When All the Hash Functions Are Weak". In: *ICALP 2008, Part II*. Ed. by L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz. Vol. 5126. LNCS. Springer, Heidelberg, July 2008, pp. 616–630. DOI: 10.1007/978-3-540-70583-3_50.

[IIM+19] A. Inoue, T. Iwata, K. Minematsu, and B. Poettering. "Cryptanalysis of OCB2: Attacks on Authenticity and Confidentiality". In: *CRYPTO 2019, Part I*. Ed. by A. Boldyreva and D. Micciancio. Vol. 11692. LNCS. Springer, Heidelberg, Aug. 2019, pp. 3–31. DOI: 10.1007/978-3-030-26948-7_1.

[IKD+08] S. Indesteege, N. Keller, O. Dunkelman, E. Biham, and B. Preneel. "A Practical Attack on KeeLoq". In: *EUROCRYPT 2008*. Ed. by N. P. Smart. Vol. 4965. LNCS. Springer, Heidelberg, Apr. 2008, pp. 1–18. DOI: 10.1007/978-3-540-78967-3_1.

[IM16] T. Iwata and K. Minematsu. "Stronger Security Variants of GCM-SIV". In: *IACR Trans. Symm. Cryptol.* 2016.1 (2016), pp. 134–157. ISSN: 2519-173X. DOI: 10.13154/tosc.v2016.i1.134-157.

[IMP+17] T. Iwata, K. Minematsu, T. Peyrin, and Y. Seurin. "ZMAC: A Fast Tweakable Block Cipher Mode for Highly Secure Message Authentication". In: *CRYPTO 2017, Part III*. Ed. by J. Katz and H. Shacham. Vol. 10403. LNCS. Springer, Heidelberg, Aug. 2017, pp. 34–65. DOI: 10.1007/978-3-319-63697-9_2.

[IMV16] T. Iwata, B. Mennink, and D. Vizár. *CENC is Optimally Secure*. Cryptology ePrint Archive, Report 2016/1087. https://eprint.iacr.org/2016/1087. 2016.

[ISO14] *On the Assessment of Cryptographic Techniques and Key Lengths, 4th edition*. ISO/IEC JTC 1/SC 27 Standing Document 12. Archived at https://web.archive.org/web/20231120102344if_/https://docplayer.net/storage/25/6483684/1700477932/FFAaPq4RA7y5QPn7TqVbew/6483684.pdf. May 2014.

[ISO19] *On the Assessment of Cryptographic Techniques and Key Lengths, 5th edition*. ISO/IEC JTC 1/SC 27 Standing Document 12. May 2019. URL: https://committee.iso.org/files/live/sites/jtc1sc27/files/resources/sc27-sd12-assessment-of-cryptographic-techniques-and-key-lengths-data.zip.

[Iwa06] T. Iwata. "New Blockcipher Modes of Operation with Beyond the Birthday Bound Security". In: *FSE 2006*. Ed. by M. J. B. Robshaw. Vol. 4047. LNCS. Springer, Heidelberg, Mar. 2006, pp. 310–327. DOI: 10.1007/11799313_20.

[JJV02] É. Jaulmes, A. Joux, and F. Valette. "On the Security of Randomized CBC-MAC Beyond the Birthday Paradox Limit: A New Construction". In: *FSE 2002*. Ed. by J. Daemen and V. Rijmen. Vol. 2365. LNCS. Springer, Heidelberg, Feb. 2002, pp. 237–251. DOI: 10.1007/3-540-45661-9_19.

[JKS+12] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. "On the Security of TLS-DHE in the Standard Model". In: *CRYPTO 2012*. Ed. by R. Safavi-Naini and R. Canetti. Vol. 7417. LNCS. Springer, Heidelberg, Aug. 2012, pp. 273–293. DOI: 10.1007/978-3-642-32009-5_17.

[Jou04] A. Joux. "Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions". In: *CRYPTO 2004*. Ed. by M. Franklin. Vol. 3152. LNCS. Springer, Heidelberg, Aug. 2004, pp. 306–316. DOI: 10.1007/978-3-540-28628-8_19.

[Jou09] A. Joux. *Algorithmic Cryptanalysis*. Chapman & Hall / CRC Press, June 2009. ISBN: 9781420070026.

[JP07] A. Joux and T. Peyrin. "Hash Functions and the (Amplified) Boomerang Attack". In: *CRYPTO 2007*. Ed. by A. Menezes. Vol. 4622. LNCS. Springer, Heidelberg, Aug. 2007, pp. 244–263. DOI: 10.1007/978-3-540-74143-5_14.

[KBP+06] J. Kim, A. Biryukov, B. Preneel, and S. Hong. "On the Security of HMAC and NMAC Based on HAVAL, MD4, MD5, SHA-0 and SHA-1 (Extended Abstract)". In: *SCN 06*. Ed. by R. D. Prisco and M. Yung. Vol. 4116. LNCS. Springer, Heidelberg, Sept. 2006, pp. 242–256. DOI: 10.1007/11832072_17.

[Kel02] J. Kelsey. "Compression and Information Leakage of Plaintext". In: *FSE 2002*. Ed. by J. Daemen and V. Rijmen. Vol. 2365. LNCS. Springer, Heidelberg, Feb. 2002, pp. 263–276. DOI: 10.1007/3-540-45661-9_21.

[Ker83] A. Kerckhoffs. "La cryptographie militaire". In: *Journal des sciences militaires* IX (Jan. 1883), pp. 5–38. URL: https://www.petitcolas.net/kerckhoffs/crypto_militaire_1_b.pdf.

[KK06] J. Kelsey and T. Kohno. "Herding Hash Functions and the Nostradamus Attack". In: *EUROCRYPT 2006*. Ed. by S. Vaudenay. Vol. 4004. LNCS. Springer, Heidelberg, May 2006, pp. 183–200. DOI: 10.1007/11761679_12.

[Kli06] V. Klima. *Tunnels in Hash Functions: MD5 Collisions Within a Minute*. Cryptology ePrint Archive, Report 2006/105. https://eprint.iacr.org/2006/105. 2006.

[KLL+16a] M. Kaplan, G. Leurent, A. Leverrier, and M. Naya-Plasencia. "Breaking Symmetric Cryptosystems Using Quantum Period Finding". In: *CRYPTO 2016, Part II*. Ed. by M. Robshaw and J. Katz. Vol. 9815. LNCS. Springer, Heidelberg, Aug. 2016, pp. 207–237. DOI: 10.1007/978-3-662-53008-5_8.

[KLL+16b] M. Kaplan, G. Leurent, A. Leverrier, and M. Naya-Plasencia. "Quantum Differential and Linear Cryptanalysis". In: *IACR Trans. Symm. Cryptol.* 2016.1 (2016), pp. 71–94. ISSN: 2519-173X. DOI: 10.13154/tosc.v2016.i1.71-94.

[KLL20] S. Kim, B. Lee, and J. Lee. "Tight Security Bounds for Double-Block Hash-then-Sum MACs". In: *EUROCRYPT 2020, Part I*. Ed. by A. Canteaut and Y. Ishai. Vol. 12105. LNCS. Springer, Heidelberg, May 2020, pp. 435–465. DOI: 10.1007/978-3-030-45721-1_16.

[KM10] H. Kuwakado and M. Morii. “Quantum distinguisher between the 3-round Feistel cipher and the random permutation”. In: *IEEE International Symposium on Information Theory, ISIT 2010*. IEEE, 2010, pp. 2682–2685. DOI: 10.1109/ISIT.2010.5513654.

[KM12] H. Kuwakado and M. Morii. “Security on the quantum-type Even-Mansour cipher”. In: *Proceedings of the International Symposium on Information Theory and its Applications, ISITA 2012*. IEEE, 2012, pp. 312–316. URL: https://ieeexplore.ieee.org/document/6400943/.

[Kob87] N. Koblitz. “Elliptic Curve Cryptosystems”. In: *Mathematics of Computation* 48.177 (1987), pp. 203–209. ISSN: 00255718, 10886842. URL: http://www.jstor.org/stable/2007884 (visited on 11/13/2023).

[KPS15] P. Karpman, T. Peyrin, and M. Stevens. “Practical Free-Start Collision Attacks on 76-step SHA-1”. In: *CRYPTO 2015, Part I*. Ed. by R. Gennaro and M. J. B. Robshaw. Vol. 9215. LNCS. Springer, Heidelberg, Aug. 2015, pp. 623–642. DOI: 10.1007/978-3-662-47989-6_30.

[KPW13] H. Krawczyk, K. G. Paterson, and H. Wee. “On the Security of the TLS Protocol: A Systematic Analysis”. In: *CRYPTO 2013, Part I*. Ed. by R. Canetti and J. A. Garay. Vol. 8042. LNCS. Springer, Heidelberg, Aug. 2013, pp. 429–448. DOI: 10.1007/978-3-642-40041-4_24.

[KR11] T. Krovetz and P. Rogaway. “The Software Performance of Authenticated-Encryption Modes”. In: *FSE 2011*. Ed. by A. Joux. Vol. 6733. LNCS. Springer, Heidelberg, Feb. 2011, pp. 306–327. DOI: 10.1007/978-3-642-21702-9_18.

[KR14] T. Krovetz and P. Rogaway. *The OCB Authenticated-Encryption Algorithm*. RFC 7253. May 2014. DOI: 10.17487/RFC7253.

[Kra03] H. Krawczyk. “SIGMA: The “SIGn-and-MAc” Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols”. In: *CRYPTO 2003*. Ed. by D. Boneh. Vol. 2729. LNCS. Springer, Heidelberg, Aug. 2003, pp. 400–425. DOI: 10.1007/978-3-540-45146-4_24.

[KS05] J. Kelsey and B. Schneier. “Second Preimages on n-Bit Hash Functions for Much Less than 2n Work”. In: *EUROCRYPT 2005*. Ed. by R. Cramer. Vol. 3494. LNCS. Springer, Heidelberg, May 2005, pp. 474–490. DOI: 10.1007/11426639_28.

[LBF08] G. Leurent, C. Bouillaguet, and P.-A. Fouque. *SIMD Is a Message Digest*. Submitted to the SHA-3 competition. 2008. URL: https://who.rocq.inria.fr/Gaetan.Leurent/simd.html.

[Leu07] G. Leurent. “Message Freedom in MD4 and MD5 Collisions: Application to APOP”. In: *FSE 2007*. Ed. by A. Biryukov. Vol. 4593. LNCS. Springer, Heidelberg, Mar. 2007, pp. 309–328. DOI: 10.1007/978-3-540-74619-5_20.

[Leu08a] G. Leurent. “MD4 is Not One-Way”. In: *FSE 2008*. Ed. by K. Nyberg. Vol. 5086. LNCS. Springer, Heidelberg, Feb. 2008, pp. 412–428. DOI: 10.1007/978-3-540-71039-4_26.

[Leu12] G. Leurent. “Analysis of Differential Attacks in ARX Constructions”. In: *ASIACRYPT 2012*. Ed. by X. Wang and K. Sako. Vol. 7658. LNCS. Springer, Heidelberg, Dec. 2012, pp. 226–243. DOI: 10.1007/978-3-642-34961-4_15.

[Leu13] G. Leurent. “Construction of Differential Characteristics in ARX Designs Application to Skein”. In: *CRYPTO 2013, Part I*. Ed. by R. Canetti and J. A. Garay. Vol. 8042. LNCS. Springer, Heidelberg, Aug. 2013, pp. 241–258. DOI: 10.1007/978-3-642-40041-4_14.

[Leu16b] G. Leurent. "Improved Differential-Linear Cryptanalysis of 7-Round Chaskey with Partitioning". In: *EUROCRYPT 2016, Part I*. Ed. by M. Fischlin and J.-S. Coron. Vol. 9665. LNCS. Springer, Heidelberg, May 2016, pp. 344–371. DOI: 10.1007/978-3-662-49890-3_14.

[LKS+06] C. Lee, J. Kim, J. Sung, S. Hong, and S. Lee. "Forgery and Key Recovery Attacks on PMAC and Mitchell's TMAC Variant". In: *ACISP 06*. Ed. by L. M. Batten and R. Safavi-Naini. Vol. 4058. LNCS. Springer, Heidelberg, July 2006, pp. 421–431. DOI: 10.1007/11780656_35.

[LMP+23] G. Leurent, B. Mennink, K. Pietrzak, and V. Rijmen. *Analysis of MinRoot: Public report*. Tech. rep. Ethereum Foundation, Sept. 2023. URL: https://crypto.ethereum.org/events/minrootanalysis2023.pdf.

[LN17a] E. List and M. Nandi. "Revisiting Full-PRF-Secure PMAC and Using It for Beyond-Birthday Authenticated Encryption". In: *CT-RSA 2017*. Ed. by H. Handschuh. Vol. 10159. LNCS. Springer, Heidelberg, Feb. 2017, pp. 258–274. DOI: 10.1007/978-3-319-52153-4_15.

[LN17b] E. List and M. Nandi. "ZMAC$^+$ – An Efficient Variable-output-length Variant of ZMAC". In: *IACR Trans. Symm. Cryptol.* 2017.4 (2017), pp. 306–325. ISSN: 2519-173X. DOI: 10.13154/tosc.v2017.i4.306-325.

[LNS18] G. Leurent, M. Nandi, and F. Sibleyras. "Generic Attacks Against Beyond-Birthday-Bound MACs". In: *CRYPTO 2018, Part I*. Ed. by H. Shacham and A. Boldyreva. Vol. 10991. LNCS. Springer, Heidelberg, Aug. 2018, pp. 306–336. DOI: 10.1007/978-3-319-96884-1_11.

[LP18] A. Luykx and B. Preneel. "Optimal Forgeries Against Polynomial-Based MACs and GCM". In: *EUROCRYPT 2018, Part I*. Ed. by J. B. Nielsen and V. Rijmen. Vol. 10820. LNCS. Springer, Heidelberg, Apr. 2018, pp. 445–467. DOI: 10.1007/978-3-319-78381-9_17.

[LP19] G. Leurent and T. Peyrin. "From Collisions to Chosen-Prefix Collisions Application to Full SHA-1". In: *EUROCRYPT 2019, Part III*. Ed. by Y. Ishai and V. Rijmen. Vol. 11478. LNCS. Springer, Heidelberg, May 2019, pp. 527–555. DOI: 10.1007/978-3-030-17659-4_18.

[LP20] G. Leurent and T. Peyrin. "SHA-1 is a Shambles: First Chosen-Prefix Collision on SHA-1 and Application to the PGP Web of Trust". In: *USENIX Security 2020*. Ed. by S. Capkun and F. Roesner. USENIX Association, Aug. 2020, pp. 1839–1856. URL: https://www.usenix.org/system/files/sec20-leurent.pdf.

[LP21] G. Leurent and C. Pernot. "New Representations of the AES Key Schedule". In: *EUROCRYPT 2021, Part I*. Ed. by A. Canteaut and F.-X. Standaert. Vol. 12696. LNCS. Springer, Heidelberg, Oct. 2021, pp. 54–84. DOI: 10.1007/978-3-030-77870-5_3.

[LPR+07] M. Lamberger, N. Pramstaller, C. Rechberger, and V. Rijmen. "Second Preimages for SMASH". In: *CT-RSA 2007*. Ed. by M. Abe. Vol. 4377. LNCS. Springer, Heidelberg, Feb. 2007, pp. 101–111. DOI: 10.1007/11967668_7.

[LPS21] G. Leurent, C. Pernot, and A. Schrottenloher. "Clustering Effect in Simon and Simeck". In: *ASIACRYPT 2021, Part I*. Ed. by M. Tibouchi and H. Wang. Vol. 13090. LNCS. Springer, Heidelberg, Dec. 2021, pp. 272–302. DOI: 10.1007/978-3-030-92062-3_10.

[LPW13] G. Leurent, T. Peyrin, and L. Wang. "New Generic Attacks against Hash-Based MACs". In: *ASIACRYPT 2013, Part II*. Ed. by K. Sako and P. Sarkar. Vol. 8270. LNCS. Springer, Heidelberg, Dec. 2013, pp. 1–20. DOI: 10.1007/978-3-642-42045-0_1.

[LS18] G. Leurent and F. Sibleyras. "The Missing Difference Problem, and Its Applications to Counter Mode Encryption". In: *EUROCRYPT 2018, Part II*. Ed. by J. B. Nielsen and V. Rijmen. Vol. 10821. LNCS. Springer, Heidelberg, Apr. 2018, pp. 745–770. DOI: 10.1007/978-3-319-78375-8_24.

[LW05] A. K. Lenstra and B. de Weger. "On the Possibility of Constructing Meaningful Hash Collisions for Public Keys". In: *ACISP 05*. Ed. by C. Boyd and J. M. G. Nieto. Vol. 3574. LNCS. Springer, Heidelberg, July 2005, pp. 267–279. DOI: 10.1007/11506157_23.

[LW15] G. Leurent and L. Wang. "The Sum Can Be Weaker Than Each Part". In: *EUROCRYPT 2015, Part I*. Ed. by E. Oswald and M. Fischlin. Vol. 9056. LNCS. Springer, Heidelberg, Apr. 2015, pp. 345–367. DOI: 10.1007/978-3-662-46800-5_14.

[Mar09] M. Marlinspike. "More tricks for defeating SSL in practice". In: *Black Hat USA* (2009). URL: https://www.blackhat.com/presentations/bh-usa-09/MARLINSPIKE/BHUSA09-Marlinspike-DefeatSSL-SLIDES.pdf.

[Mat94] M. Matsui. "Linear Cryptanalysis Method for DES Cipher". In: *EUROCRYPT'93*. Ed. by T. Helleseth. Vol. 765. LNCS. Springer, Heidelberg, May 1994, pp. 386–397. DOI: 10.1007/3-540-48285-7_33.

[McE78] R. J. McEliece. "A Public-Key Cryptosystem Based On Algebraic Coding Theory". In: *Deep Space Network Progress Report* 44 (Jan. 1978), pp. 114–116. URL: https://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF.

[McG12] D. McGrew. *Impossible plaintext cryptanalysis and probable-plaintext collision attacks of 64-bit block cipher modes*. Cryptology ePrint Archive, Report 2012/623. Accepted to FSE 2013. Nov. 2012. URL: https://eprint.iacr.org/2012/623.

[MDK14] B. Möller, T. Duong, and K. Kotowicz. *This POODLE Bites: Exploiting The SSL 3.0 Fallback*. 2014. URL: https://www.openssl.org/~bodo/ssl-poodle.pdf.

[Mer90] R. C. Merkle. "One Way Hash Functions and DES". In: *CRYPTO'89*. Ed. by G. Brassard. Vol. 435. LNCS. Springer, Heidelberg, Aug. 1990, pp. 428–446. DOI: 10.1007/0-387-34805-0_40.

[Mil86] V. S. Miller. "Use of Elliptic Curves in Cryptography". In: *CRYPTO'85*. Ed. by H. C. Williams. Vol. 218. LNCS. Springer, Heidelberg, Aug. 1986, pp. 417–426. DOI: 10.1007/3-540-39799-X_31.

[MLM+14] K. Minematsu, S. Lucks, H. Morita, and T. Iwata. "Attacks and Security Proofs of EAX-Prime". In: *FSE 2013*. Ed. by S. Moriai. Vol. 8424. LNCS. Springer, Heidelberg, Mar. 2014, pp. 327–347. DOI: 10.1007/978-3-662-43933-3_17.

[Moz23] Mozilla. *Measurement Dashboard*. Accessed on 2023-11-08. Nov. 2023. URL: https://telemetry.mozilla.org/new-pipeline/dist.html#!measure=SSL_CIPHER_SUITE_FULL&max_channel_version=nightly%252F121&min_channel_version=nightly%252F121.

[MRS15] F. Mendel, V. Rijmen, and M. Schläffer. "Collision Attack on 5 Rounds of Grøstl". In: *FSE 2014*. Ed. by C. Cid and C. Rechberger. Vol. 8540. LNCS. Springer, Heidelberg, Mar. 2015, pp. 509–521. DOI: 10.1007/978-3-662-46706-0_26.

[MS02] I. Mantin and A. Shamir. "A Practical Attack on Broadcast RC4". In: *FSE 2001*. Ed. by M. Matsui. Vol. 2355. LNCS. Springer, Heidelberg, Apr. 2002, pp. 152–164. DOI: 10.1007/3-540-45473-X_13.

[MV04] D. A. McGrew and J. Viega. "The Security and Performance of the Galois/Counter Mode (GCM) of Operation". In: *INDOCRYPT 2004*. Ed. by A. Canteaut and K. Viswanathan. Vol. 3348. LNCS. Springer, Heidelberg, Dec. 2004, pp. 343–355. DOI: 10.1007/978-3-540-30556-9_27.

[Nai15] Y. Naito. "Full PRF-Secure Message Authentication Code Based on Tweakable Block Cipher". In: *ProvSec 2015*. Ed. by M. H. Au and A. Miyaji. Vol. 9451. LNCS. Springer, Heidelberg, Nov. 2015, pp. 167–182. DOI: 10.1007/978-3-319-26059-4_9.

[Nai17] Y. Naito. "Blockcipher-Based MACs: Beyond the Birthday Bound Without Message Length". In: *ASIACRYPT 2017, Part III*. Ed. by T. Takagi and T. Peyrin. Vol. 10626. LNCS. Springer, Heidelberg, Dec. 2017, pp. 446–470. DOI: 10.1007/978-3-319-70700-6_16.

[Nai18] Y. Naito. "Improved Security Bound of LightMAC_Plus and Its Single-Key Variant". In: *CT-RSA 2018*. Ed. by N. P. Smart. Vol. 10808. LNCS. Springer, Heidelberg, Apr. 2018, pp. 300–318. DOI: 10.1007/978-3-319-76953-0_16.

[NBS80] National Bureau of Standards. *DES Modes of Operation*. FIPS 81. Dec. 1980. DOI: 10.6028/NBS.FIPS.81.

[NBS85] National Bureau of Standards. *Computer Data Authentication*. FIPS 113. May 1985. DOI: 10.6028/NBS.FIPS.113.

[NIS01] M. Dworkin. *Recommendation for Block Cipher Modes of Operation. Methods and Techniques*. NIST Special Publication 800-38A. National Institute for Standards and Technology, Dec. 2001. DOI: 10.6028/NIST.SP.800-38A.

[NIS12] W. C. Barker and E. Barker. *Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher*. NIST Special Publication 800-67. Revision 1. National Institute for Standards and Technology, Jan. 2012. DOI: 10.6028/NIST.SP.800-67r1.

[NIS17] E. Barker and N. Mouha. *Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher*. NIST Special Publication 800-67. Revision 2. National Institute for Standards and Technology, Nov. 2017. DOI: 10.6028/NIST.SP.800-67r2.

[NIS19] E. Barker and A. Roginsky. *Transitioning the Use of Cryptographic Algorithms and Key Lengths*. NIST Special Publication 800-131A. Revision 2. National Institute for Standards and Technology, Mar. 2019. DOI: 10.6028/NIST.SP.800-131Ar2.

[NKB06] C. Namprempre, T. Kohno, and M. Bellare. *The Secure Shell (SSH) Transport Layer Encryption Modes*. RFC 4344. Jan. 2006. DOI: 10.17487/RFC4344.

[PC15] G. Procter and C. Cid. "On Weak Keys and Forgery Attacks Against Polynomial-Based MAC Schemes". In: *Journal of Cryptology* 28.4 (Oct. 2015), pp. 769–795. DOI: 10.1007/s00145-014-9178-9.

[Pey07] T. Peyrin. "Cryptanalysis of Grindahl". In: *ASIACRYPT 2007*. Ed. by K. Kurosawa. Vol. 4833. LNCS. Springer, Heidelberg, Dec. 2007, pp. 551–567. DOI: 10.1007/978-3-540-76900-2_34.

[Pol75] J. M. Pollard. “A Monte Carlo method for factorization”. In: *BIT Numerical Mathematics* 15.3 (Sept. 1975), pp. 331–334. ISSN: 1572-9125. DOI: 10.1007/BF01933667.

[PR00] E. Petrank and C. Rackoff. “CBC MAC for Real-Time Data Sources”. In: *Journal of Cryptology* 13.3 (June 2000), pp. 315–338. DOI: 10.1007/s001450010009.

[Pv95] B. Preneel and P. C. van Oorschot. “MDx-MAC and Building Fast MACs from Hash Functions”. In: *CRYPTO’95*. Ed. by D. Coppersmith. Vol. 963. LNCS. Springer, Heidelberg, Aug. 1995, pp. 1–14. DOI: 10.1007/3-540-44750-4_1.

[Pv96] B. Preneel and P. C. van Oorschot. “On the Security of Two MAC Algorithms”. In: *EUROCRYPT’96*. Ed. by U. M. Maurer. Vol. 1070. LNCS. Springer, Heidelberg, May 1996, pp. 19–32. DOI: 10.1007/3-540-68339-9_3.

[PW14] T. Peyrin and L. Wang. “Generic Universal Forgery Attack on Iterative Hash-Based MACs”. In: *EUROCRYPT 2014*. Ed. by P. Q. Nguyen and E. Oswald. Vol. 8441. LNCS. Springer, Heidelberg, May 2014, pp. 147–164. DOI: 10.1007/978-3-642-55220-5_9.

[RD12] J. Rizzo and T. Duong. “The CRIME attack”. In: *EKOparty Security Conference*. Vol. 2012. 2012. URL: https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit.

[Reg05] O. Regev. “On lattices, learning with errors, random linear codes, and cryptography”. In: *37th ACM STOC*. Ed. by H. N. Gabow and R. Fagin. ACM Press, May 2005, pp. 84–93. DOI: 10.1145/1060590.1060603.

[RFC1321] R. L. Rivest. *The MD5 Message-Digest Algorithm*. RFC 1321. Apr. 1992. DOI: 10.17487/RFC1321.

[RFC2246] C. Allen and T. Dierks. *The TLS Protocol Version 1.0*. RFC 2246. Jan. 1999. DOI: 10.17487/RFC2246.

[RFC4270] P. E. Hoffman and B. Schneier. *Attacks on Cryptographic Hashes in Internet Protocols*. RFC 4270. Dec. 2005. DOI: 10.17487/RFC4270.

[RFC4880] H. Finney, L. Donnerhacke, J. Callas, R. L. Thayer, and D. Shaw. *OpenPGP Message Format*. RFC 4880. Nov. 2007. DOI: 10.17487/RFC4880.

[RFC4894] P. E. Hoffman. *Use of Hash Algorithms in Internet Key Exchange (IKE) and IPsec*. RFC 4894. May 2007. DOI: 10.17487/RFC4894.

[RFC8996] K. Moriarty and S. Farrell. *Deprecating TLS 1.0 and TLS 1.1*. RFC 8996. Mar. 2021. DOI: 10.17487/RFC8996.

[RFC9155] L. Velvindron, K. Moriarty, and A. Ghedini. *Deprecating MD5 and SHA-1 Signature Hashes in TLS 1.2 and DTLS 1.2*. RFC 9155. Dec. 2021. DOI: 10.17487/RFC9155.

[Riv91] R. L. Rivest. “The MD4 Message Digest Algorithm”. In: *CRYPTO’90*. Ed. by A. J. Menezes and S. A. Vanstone. Vol. 537. LNCS. Springer, Heidelberg, Aug. 1991, pp. 303–311. DOI: 10.1007/3-540-38424-3_22.

[Rog04] P. Rogaway. “Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC”. In: *ASIACRYPT 2004*. Ed. by P. J. Lee. Vol. 3329. LNCS. Springer, Heidelberg, Dec. 2004, pp. 16–31. DOI: 10.1007/978-3-540-30539-2_2.

[Rog11] P. Rogaway. *Evaluation of Some Blockcipher Modes of Operation*. Tech. rep. CRYPREC, Feb. 2011. URL: https://web.cs.ucdavis.edu/~rogaway/papers/modes.pdf.

[RR08] C. Rechberger and V. Rijmen. "New Results on NMAC/HMAC when Instantiated with Popular Hash Functions". In: *J. Univers. Comput. Sci.* 14.3 (2008), pp. 347–376. DOI: 10.3217/JUCS-014-03-0347.

[RSA78] R. L. Rivest, A. Shamir, and L. M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Communications of the Association for Computing Machinery* 21.2 (Feb. 1978), pp. 120–126. DOI: 10.1145/359340.359342.

[Saa12] M.-J. O. Saarinen. "Cycling Attacks on GCM, GHASH and Other Polynomial MACs and Hashes". In: *FSE 2012*. Ed. by A. Canteaut. Vol. 7549. LNCS. Springer, Heidelberg, Mar. 2012, pp. 216–225. DOI: 10.1007/978-3-642-34047-5_13.

[SBK+17] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. "The First Collision for Full SHA-1". In: *CRYPTO 2017, Part I*. Ed. by J. Katz and H. Shacham. Vol. 10401. LNCS. Springer, Heidelberg, Aug. 2017, pp. 570–596. DOI: 10.1007/978-3-319-63688-7_19.

[SHA1] National Institute for Standards and Technology. *Secure Hash Standard*. FIPS 180-1. Apr. 1995. DOI: 10.6028/NIST.FIPS.180-1.

[SHA2] National Institute for Standards and Technology. *Secure Hash Standard*. FIPS 180-2. Aug. 2002. URL: https://csrc.nist.gov/files/pubs/fips/180-2/final/docs/fips180-2.pdf.

[SHA3] National Institute for Standards and Technology. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. FIPS 202. Aug. 2015. DOI: 10.6028/NIST.FIPS.202.

[Sha49] C. E. Shannon. "Communication theory of secrecy systems". In: *Bell Systems Technical Journal* 28.4 (1949), pp. 656–715. DOI: 10.1002/j.1538-7305.1949.tb00928.x.

[Sho94] P. W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *35th FOCS*. IEEE Computer Society Press, Nov. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[Sho96] V. Shoup. "On Fast and Provably Secure Message Authentication Based on Universal Hashing". In: *CRYPTO'96*. Ed. by N. Koblitz. Vol. 1109. LNCS. Springer, Heidelberg, Aug. 1996, pp. 313–328. DOI: 10.1007/3-540-68697-5_24.

[Sho97] P. W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM J. Comput.* 26.5 (1997), pp. 1484–1509. DOI: 10.1137/S0097539795293172.

[Sim97] D. R. Simon. "On the Power of Quantum Computation". In: *SIAM J. Comput.* 26.5 (1997), pp. 1474–1483. DOI: 10.1137/S0097539796298637.

[SLW07] M. Stevens, A. K. Lenstra, and B. de Weger. "Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities". In: *EUROCRYPT 2007*. Ed. by M. Naor. Vol. 4515. LNCS. Springer, Heidelberg, May 2007, pp. 1–22. DOI: 10.1007/978-3-540-72540-4_1.

[SSA+09] M. Stevens, A. Sotirov, J. Appelbaum, A. K. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger. "Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate". In: *CRYPTO 2009*. Ed. by S. Halevi. Vol. 5677. LNCS. Springer, Heidelberg, Aug. 2009, pp. 55–69. DOI: 10.1007/978-3-642-03356-8_4.

[Sta22a] Statista. *Data volume of global consumer IP traffic from 2017 to 2022*. Accessed on 2023-11-21. 2022. URL: https://www.statista.com/statistics/267202/global-data-volume-of-consumer-ip-traffic/.

[Sta22b] Statista. *Total installed based of data storage capacity in the global datasphere from 2020 to 2025*. Accessed on 2023-11-21. 2022. URL: https://www.statista.com/statistics/1185900/worldwide-datasphere-storage-capacity-installed-base/.

[Ste13] M. Stevens. "New Collision Attacks on SHA-1 Based on Optimal Joint Local-Collision Analysis". In: *EUROCRYPT 2013*. Ed. by T. Johansson and P. Q. Nguyen. Vol. 7881. LNCS. Springer, Heidelberg, May 2013, pp. 245–261. DOI: 10.1007/978-3-642-38348-9_15.

[Ste99] F. A. Stevenson. *Cryptanalysis of Contents Scrambling System*. Online. Nov. 1999. URL: https://insecure.org/news/cryptanalysis_of_contents_scrambling_system.htm.

[SY17] F. Song and A. Yun. "Quantum Security of NMAC and Related Constructions - PRF Domain Extension Against Quantum attacks". In: *CRYPTO 2017, Part II*. Ed. by J. Katz and H. Shacham. Vol. 10402. LNCS. Springer, Heidelberg, Aug. 2017, pp. 283–309. DOI: 10.1007/978-3-319-63715-0_10.

[Tsu92] G. Tsudik. "Message Authentication with One-way Hash Functions". In: *SIGCOMM Comput. Commun. Rev.* 22.5 (Oct. 1992), pp. 29–38. ISSN: 0146-4833. DOI: 10.1145/141809.141812.

[TWP08] E. Tews, R.-P. Weinmann, and A. Pyshkin. "Breaking 104 Bit WEP in Less Than 60 Seconds". In: *WISA 07*. Ed. by S. Kim, M. Yung, and H.-W. Lee. Vol. 4867. LNCS. Springer, Heidelberg, Aug. 2008, pp. 188–202. DOI: 10.1007/978-3-540-77535-5_14.

[Vau02] S. Vaudenay. "Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS..." In: *EUROCRYPT 2002*. Ed. by L. R. Knudsen. Vol. 2332. LNCS. Springer, Heidelberg, Apr. 2002, pp. 534–546. DOI: 10.1007/3-540-46035-7_35.

[VP15] M. Vanhoef and F. Piessens. "All Your Biases Belong to Us: Breaking RC4 in WPA-TKIP and TLS". In: *USENIX Security 2015*. Ed. by J. Jung and T. Holz. USENIX Association, Aug. 2015, pp. 97–112. URL: https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-vanhoef.pdf.

[vW94] P. C. van Oorschot and M. J. Wiener. "Parallel Collision Search with Application to Hash Functions and Discrete Logarithms". In: *ACM CCS 94*. Ed. by D. E. Denning, R. Pyle, R. Ganesan, and R. S. Sandhu. ACM Press, Nov. 1994, pp. 210–218. DOI: 10.1145/191177.191231.

[vW99] P. C. van Oorschot and M. J. Wiener. "Parallel Collision Search with Cryptanalytic Applications". In: *Journal of Cryptology* 12.1 (Jan. 1999), pp. 1–28. DOI: 10.1007/PL00003816.

[WC81] M. N. Wegman and L. Carter. "New Hash Functions and Their Use in Authentication and Set Equality". In: *Journal of Computer and System Sciences* 22 (1981), pp. 265–279. DOI: 10.1016/0022-0000(81)90033-7.

[WHW+23] P. Wouters, D. Huigens, J. Winter, and N. Yutaka. *OpenPGP*. Internet-Draft draft-ietf-openpgp-crypto-refresh-12. Work in Progress. Oct. 2023. 199 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-openpgp-crypto-refresh/12/.

[WLF+05] X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu. "Cryptanalysis of the Hash Functions MD4 and RIPEMD". In: *EUROCRYPT 2005*. Ed. by R. Cramer. Vol. 3494. LNCS. Springer, Heidelberg, May 2005, pp. 1–18. DOI: 10.1007/11426639_1.

[WMA+19] L. Wouters, E. Marin, T. Ashur, B. Gierlichs, and B. Preneel. "Fast, Furious and Insecure: Passive Keyless Entry and Start Systems in Modern Supercars". In: *IACR TCHES* 2019.3 (2019), pp. 66–85. ISSN: 2569-2925. DOI: 10.13154/tches.v2019.i3.66-85.

[WY05] X. Wang and H. Yu. "How to Break MD5 and Other Hash Functions". In: *EUROCRYPT 2005*. Ed. by R. Cramer. Vol. 3494. LNCS. Springer, Heidelberg, May 2005, pp. 19–35. DOI: 10.1007/11426639_2.

[WYW+09] X. Wang, H. Yu, W. Wang, H. Zhang, and T. Zhan. "Cryptanalysis on HMAC/NMAC-MD5 and MD5-MAC". In: *EUROCRYPT 2009*. Ed. by A. Joux. Vol. 5479. LNCS. Springer, Heidelberg, Apr. 2009, pp. 121–133. DOI: 10.1007/978-3-642-01001-9_7.

[WYY05] X. Wang, Y. L. Yin, and H. Yu. "Finding Collisions in the Full SHA-1". In: *CRYPTO 2005*. Ed. by V. Shoup. Vol. 3621. LNCS. Springer, Heidelberg, Aug. 2005, pp. 17–36. DOI: 10.1007/11535218_2.

[Yas07] K. Yasuda. ""Sandwich" Is Indeed Secure: How to Authenticate a Message with Just One Hashing". In: *ACISP 07*. Ed. by J. Pieprzyk, H. Ghodosi, and E. Dawson. Vol. 4586. LNCS. Springer, Heidelberg, July 2007, pp. 355–369. DOI: 10.1007/978-3-540-73458-1_26.

[Yas10] K. Yasuda. "The Sum of CBC MACs Is a Secure PRF". In: *CT-RSA 2010*. Ed. by J. Pieprzyk. Vol. 5985. LNCS. Springer, Heidelberg, Mar. 2010, pp. 366–381. DOI: 10.1007/978-3-642-11925-5_25.

[Yas11] K. Yasuda. "A New Variant of PMAC: Beyond the Birthday Bound". In: *CRYPTO 2011*. Ed. by P. Rogaway. Vol. 6841. LNCS. Springer, Heidelberg, Aug. 2011, pp. 596–609. DOI: 10.1007/978-3-642-22792-9_34.

[ZWS+12] L. Zhang, W. Wu, H. Sui, and P. Wang. "3kf9: Enhancing 3GPP-MAC beyond the Birthday Bound". In: *ASIACRYPT 2012*. Ed. by X. Wang and K. Sako. Vol. 7658. LNCS. Springer, Heidelberg, Dec. 2012, pp. 296–312. DOI: 10.1007/978-3-642-34961-4_19.

Selected Publications

The Sum Can Be Weaker Than Each Part*

Gaëtan Leurent[1] and Lei Wang[2]

[1] Inria, France, gaetan.leurent@inria.fr
[2] Nanyang Technological University, Singapore, wang.lei@ntu.edu.sg

Abstract. In this paper we study the security of summing the outputs of two *independent* hash functions, in an effort to increase the security of the resulting design, or to hedge against the failure of one of the hash functions. The exclusive-or (XOR) combiner $H_1(M) \oplus H_2(M)$ is one of the two most classical combiners, together with the concatenation combiner $H_1(M) \parallel H_2(M)$. While the security of the concatenation of two hash functions is well understood since Joux's seminal work on multicollisions, the security of the sum of two hash functions has been much less studied. The XOR combiner is well known as a good PRF and MAC combiner, and is used in practice in TLS versions 1.0 and 1.1. In a hash function setting, Hoch and Shamir have shown that if the compression functions are modeled as random oracles, or even *weak* random oracles (*i.e.* they can easily be inverted – in particular H_1 and H_2 offer no security), $H_1 \oplus H_2$ is indifferentiable from a random oracle up to the birthday bound.

In this work, we focus on the preimage resistance of the sum of two narrow-pipe n-bit hash functions, following the Merkle-Damgård or HAIFA structure (the internal state size and the output size are both n bits). We show a rather surprising result: the sum of two such hash functions, e.g. SHA-512 $\oplus$ Whirlpool, can never provide n-bit security for preimage resistance. More precisely, we present a generic preimage attack with a complexity of $\tilde{O}(2^{5n/6})$. While it is already known that the XOR combiner is not preserving for preimage resistance (*i.e.* there might be *some* instantiations where the hash functions are secure but the sum is not), our result is much stronger: for *any* narrow-pipe functions, the sum is not preimage resistant.

Besides, we also provide concrete preimage attacks on the XOR combiner (and the concatenation combiner) when one or both of the compression functions are weak; this complements Hoch and Shamir's proof by showing its tightness for preimage resistance.

Of independent interests, one of our main technical contributions is a novel structure to control *simultaneously* the behavior of independent hash computations which share the same input message. We hope that breaking the pairwise relationship between their internal states will have applications in related settings.

Keywords: Hash functions, combiners, XOR combiner, preimage attack.

1 Introduction

Hash functions are a very important class of primitive in modern cryptography, used in almost every secure system. A hash function $H : \{0,1\}^* \mapsto \{0,1\}^n$ takes an arbitrary length input and produces an n-bit output or digest. Hash functions are used in many setting with various security requirements; the general expectation is that a hash function should behave like a random function from $\{0,1\}^*$ to $\{0,1\}^n$. More concretely, the main security notions expected from a hash function are:

Collision resistance. It should be hard to find two messages $M \neq M'$ with $H(M) = H(M')$.

Second-preimage resistance. Given a message M, it should be hard to find $M' \neq M$ with $H(M) = H(M')$.

Preimage resistance. Given a target hash value $\overline{H}$, it should be hard to find M with $H(M) = \overline{H}$.

Since generic collision attacks require $2^{n/2}$ work, and generic preimage attacks require 2^n work, a secure hash function should have the same level of resistance.

In order to build more secure hash functions, or to protect oneself against future cryptanalysis advances, such as the devastating attacks of Wang *et al.* against the SHA family [42,41], a practical countermeasure might be to combine two different hash functions. The goal is that the combined hash function can only be broken when both components are weak. In particular, this reasoning was used by the designers of SSL [14] and TLS [8], who combined MD5 and SHA-1 in various ways. More precisely, the Key Derivation Function of TLS v1.0/v1.1 uses a sum of HMAC-MD5 and HMAC-SHA-1.[3] The designers explain [8]: "In order to make the PRF as secure as possible, it uses two hash algorithms in a way which should guarantee its security if either algorithm remains secure."

There are two classical hash function combiners: the concatenation combiner $H_1(M) \,\|\, H_2(M)$ and the XOR combiner $H_1(M) \oplus H_2(M)$. In a seminal work [20], Joux showed that the concatenation combiner with narrow-pipe hash functions offers much less security than could be expected: it has $2n$ bits of output, but essentially offers the same security as an n-bit hash function. In this work, we carry a similar analysis for the XOR combiner. Previous work has shown that it is indifferentiable from a random oracle up to the birthday bound [17], even if the initial functions are weak; in particular, it has optimal collision resistance of $n/2$ bits. However, we show that the preimage security is much less than one might expect, with a generic preimage attack with complexity $\tilde{O}(2^{5n/6})$.

Since the goal of a combiner is to keep some security even if one of the functions is found to be weak, it is natural that the two hash functions H_1 and H_2 are independent in practice. Throughout this paper the two hash functions

[3] We note that this MD5/SHA-1 combiner has been replaced by primitives based on single hash function (e.g., SHA-256) since TLS v1.2 [9].

used in a combiner are always assumed to be independent without specifying it explicitly[4].

Iterated hash function. In this paper we consider iterated hash functions, following the Merkle-Damgård construction [30,7] or the more general HAIFA construction [3], as shown in Figure 1. We focus on narrow-pipe designs, *i.e.* we assume that the internal state size is the same as the output size n. The message M is first split into blocks $m_0, \dots m_\ell$, and the hash function iterates a series of compression functions h_i over an internal state x, with the initial value denoted as IV. Finally, the hash value is computed with a finalization function g:

$$x_0 = IV \qquad x_{i+1} = h_i(x_i, m_i) \qquad H(M) = g(x_{\ell+1}, |M|)$$

In the following, we assume that the compression function is the same at every step ($\forall i, h_i = h$) in order to simplify the notations, but it is straightforward to apply the attack with the corresponding function at each step.

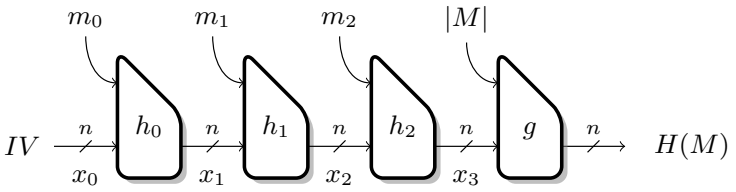


Fig. 1. Iterated hash function.

1.1 Related works

Combiners have been studied in several different settings. For generic attacks, the compression functions are modeled as random functions, in order to devise attacks that don't use any weakness of the compression functions. On the other hand, some work assumes that the compression function is a *weak* random oracle (that can easily be inverted), and prove that some constructions are still secure in this model. Finally, more theoretical work focus on the notion of *robustness*, *i.e.* the existence of a reduction from attacks on the hash functions to attacks on the combiner.

[4] If the two hash functions can be related, it is trivial that the XOR combiner is not security-preserving. For instance, let $H_2(M) := H_1(M) \oplus \texttt{const}$, where `const` is a constant. If H_1 is ideally secure, then H_2 is also ideally secure. However, the XOR combiner $H_1(M) \oplus H_2(M) = \texttt{const}$ for any message M.

Analysis of the concatenation combiner. The concatenation combiner $H_1(M) \| H_2(M)$ is probably the most studied one. In 2004, Joux [20] described surprising attacks on the concatenation of two narrow-pipe hash functions using multicollisions: while the output size is $2n$ bits, the concatenation can at most provide $n/2$-bit security for collision resistance and n-bit security for preimage resistance[5]. In particular, the concatenation is not security-amplifying. On the other hand, the concatenation combiner is robust for preimages and collisions, which gives a matching lower bound for generic attacks.

Later, Hoch and Shamir [17] evaluated the security of the concatenation combiner with two weak hash functions. More precisely, the two hash functions are narrow-pipe Merkle-Damgård, and the compression functions are modeled as weak random oracles (as defined by Liskov [24]), *i.e.*, the adversary is given additional interfaces to receive (random) preimages of the compression functions. They have proven that in this model, the concatenation combiner is still indifferentiable from a random oracle with $n/2$-bit security, implying (at least) the same security bound for collision resistance and preimage resistance. The bound is matched by Joux's attack for collisions, but there is a gap with Joux's attack for preimages, with complexity 2^n, which might be interesting to investigate further.

Mendel *et al.* analyzed some dedicated instantiations of the concatenation combiner [28], in particular using the hash function MD5. We omit the details and refer interested readers to [28].

Analysis of the XOR combiner. The XOR combiner has received less analysis. The work of Hoch and Shamir [17] actually proves the security of the XOR combiner as an intermediate result: it is also indifferentiable from a random oracle up to $2^{n/2}$ queries in the weak random oracle model. In particular, this proves that there are no generic attacks with complexity smaller than $2^{n/2}$. For collision resistance, the bound is tight, since it is matched with the generic birthday attack bound. On the other hand, for preimage resistance, there exists a gap between the $n/2$-bit proven bound and the n-bit expected ideal security bound.

To the best of our knowledge, no preimage attacks have been shown against the XOR combiner. Therefore, the preimage security of the XOR combiner against generic attacks is still an open problem, and will be the main topic of our work. We will also consider the preimage security of the XOR combiner with weak hash functions, and study the tightness of Hoch and Shamir's bound.

Robust combiners. In the last years, the general problem of combining two (or more) hash functions H_1 and H_2 has been extensively studied from a theoretical point of view. These works focus on the notion of a *robust* combiner: a robust combiner is secure with respect to property α as long as one of the underlying hash functions is secure for α. It can be shown that the concatenation combiner is a robust combiner for collision-resistant hash functions and for MACs, while the

[5] The attacks actually require only one of the functions to be narrow-pipe.

XOR combiner is robust for PRFs and for MACs [23]. More advanced combiners have been constructed in order to be robust for multiple properties simultaneously [11,12,13]. The notion was mostly studied via the black-box reduction model. A series of results have showed that robust combiners for collision resistance and preimage resistance cannot have an output length significantly shorter than the sum of the output length of the underlying hash functions [4,35,36,31]. Since the XOR combiner is length preserving, this shows that it is not robust for collision resistance and preimage resistance.

Actually, the impossibility results are in part due to the stringent requirement from the black-box reduction model. In order to overcome this limitation, Mittelbach introduced the *idealized* random oracle model [32]. He gives a construction of a short output combiner with optimal collision and preimage security in this model[6] (assuming that one of the functions is ideal): Cryptophia's short combiner uses the sum of two hash functions with some pre-processing of the messages (to allow non-independent functions).

More generally, we point out that a combiner being non-robust does not necessarily mean there is an attack. The non-robustness results only show that the security of the combiner cannot be proved with a *reduction* from the security of the hash functions. In particular, the XOR combiner is not robust for collision-resistance, or even collision-resistance preserving. However, Hoch and Shamir's work proves that there are no generic collision attacks on this construction, either with ideal compression function, or even with weak compression functions. This arguably makes the XOR a useful combiner for collision resistance. Regarding preimage security, the non-robustness result does not imply that the XOR of two concrete hash functions is weak, and the simplicity and short output of this construction still make it quite attractive.

1.2 Our results

In this work, we study the preimage security of the XOR combiner, the main remaining open problem for classical combiners. We show that, surprisingly, the sum of two narrow-pipe n-bit hash functions can never achieve n-bit security for preimage resistance. More precisely, we find a generic preimage attack with a complexity of $\tilde{O}(2^{5n/6})$. It does not exploit any structural weakness of the compression functions and hence is applicable even if the compression functions are two ideal random oracles. Thus, even if the two hash functions are n-bit secure for preimage resistance, the XOR combiner is at most $5n/6$-bit secure for primage resistance. In other words, *the sum can be weaker than each part.*

The attack is based on a novel technique to break the pairwise relationship between the internal states of the two hash functions. More precisely, the two hash functions H_1 and H_2 share the same input message, and hence the internal states of their iterative compression function computations are related. We control the computation chains of H_1 and H_2 simultaneously by constructing a new message

[6] A mistake in the initial proof and construction was later fixed by Mennink and Preneel [29].

structure $\mathcal{M}$, and two sets of internal states $\mathcal{A}$ for H_1 and $\mathcal{B}$ for H_2 such that: for any value A from $\mathcal{A}$ and any value B from $\mathcal{B}$, we can derive a message $M_{A,B}$ from $\mathcal{M}$ such that $H_1(M_{A,B})$ produces A and $H_2(M_{A,B})$ produces B. Hence we can select states from $\mathcal{A}$ and $\mathcal{B}$ independently. After that, we use a birthday match to find a message block m, a value A from $\mathcal{A}$ and a value B from $\mathcal{B}$ such that $h_1(A, m) \oplus h_2(B, m)$ is equal to the target hash digest, where h_1 and h_2 are the compression functions of H_1 and H_2 respectively. Finally we derive the message $M_{A,B}$ from $\mathcal{M}$, and output $M_{A,B} \parallel m$ as a preimage of the target hash digest.

Our preimage attack is also applicable to Cryptophia's short combiner [32,29]. This construction has been proven to provide optimal collision and preimage resistance, assuming that at least one of the initial functions is a monolithic random oracle, but our attack does not violate the security proof, because we use the fact that both functions have an iterative structures with an n-bit internal state. Still, this shows that with many practical hash functions, the combiner will be weaker that the initial functions. Our results also show that the XOR combiner and Cryptophia's combiner are not robust in the semi-black-box model introduced by Mittelbach [32][7], even with independant hash functions H_1 and H_2.

Our analysis on the XOR combiner is also interesting for dedicated hash function design. The hash function family RIPEMD [10] is based on a compression function with two parallel lanes, added together at the end of each compression function. Interestingly, RIPEMD-160 has been quite resilient to cryptanalysis [27,38,25,26], and are still considered secure. Several more recent designs use parallel lanes in a similar way (combining them at the end of each compression function call), such as HAS-V [34], FORK [18] and LANE [19]. It might be tempting to use parallel lanes during the full iteration, and to combine them only at the end. Indeed, the designers of SHA-V [15] used this approach: the 160-bit version of SHA-V has two parallel lanes, combined at the end with a modular sum. This is equivalent to summing two different hash functions, and hence our attack can be applied to SHA-V.

Another contribution of this paper is to present concrete preimage attacks on the XOR combiner with one or both weak hash functions (defined in [24]). The complexity of our attacks is $\tilde{O}(2^{n/2})$. Furthermore, the attack can be extended to the concatenation combiner with two weak hash functions under the same complexity. It can be seen that these attacks match the bound of Hoch and Shamir's security proof [17], and hence fulfill the gaps pointed out in Section 1.1. It implies the tightness of Hoch and Shamir's proof on the classical combiners with weak hash functions for preimage resistance.

Finally, we would like to highlight the technical interests of this paper. We devise a novel structure named interchange structure to simultaneously control

[7] Loosely speaking, a combiner is robust with respect to property α if it is (at least) as secure as the stronger underlying hash function for α.

two (or more) hash lanes with the same input message, and succeed in further relaxing the pairwise relation between the internal states of lanes. It is indeed a step of technical advance compared with previous extensive studies on this topic, and hence will hopefully have applications or lead to new technical development in related settings. We refer to the open discussions in Section 7 for more details.

1.3 Notations and roadmap in the rest of paper

We use the following notations:

$H_1,\ H_2$	: hash functions
IV_1, IV_2	: initial values for H_1 and H_2, respectively
$h_1,\ h_2$	: compression functions of H_1 and H_2, respectively
$h_1^*,\ h_2^*$	: compression functions iterated over several blocks (in particular, $H_i(\boldsymbol{M}) = h_i^*(IV_i, \boldsymbol{M})$)
m	: message block
M	: message chunk ($n/2$ blocks)
$\boldsymbol{M}$	: long message (several chunks)
$a_j,\ b_k$	: chains for H_1 and H_2, respectively; a_j denotes a generic chain, while a_{j_0} denotes a particular chain
$A_j,\ B_k$	: end points of the chains
n	: hash function output size

Roadmap. Section 2 provides an overview of our generic preimage attack on the XOR combiner. Sections 3, 4, and 5 elaborate the attack procedure step by step in details. Section 6 presents the applications and extensions of the attack. Finally we conclude the paper and discuss future directions in Section 7.

2 Overview of the attack

We first give an overview of the techniques and the structures used in the attack, while more detailed descriptions will be given in the following sections.

The main idea is to consider several chains of internal states reached by processing a common message $\boldsymbol{M}$ from different starting points (note that the message $\boldsymbol{M}$ is not fixed in advance, but will be determined when building the structure). More precisely, the message $\boldsymbol{M}$ is denoted as the *primary* message, and divided in several chunks: $\boldsymbol{M} = M_0 \,\|\, M_1 \,\|\, \ldots$ (as discussed later, a chunk will consist of several message blocks). We denote chains of internal states for H_1 as a_j, and the individual states of the chain as a_j^i, with $h_1^*(a_j^i, M_i) = a_j^{i+1}$. Similarly, we denote chains for H_2 as b_k, with $h_2^*(b_k^i, M_i) = b_k^{i+1}$. When considering both hash functions, message block M_i leads from the pair of states (a_j^i, b_k^i) to (a_j^{i+1}, b_k^{i+1}), which is denoted:

$$(a_j^i, b_k^i) \overset{M_i}{\rightsquigarrow} (a_j^{i+1}, b_k^{i+1}).$$

Switch structure. Next we build special structures called *switches* in order to jump between chains in a controlled way. A switch allows to jump from a specific pair of chains (a_{j_0}, b_{k_0}) to a different pair of chains (a_{j_0}, b_{k_1}) using a secondary message chunk M'_i, in addition to the normal transitions using chunk M_i of the primary message $\boldsymbol{M}$:

$$(a^i_j, b^i_k) \overset{M_i}{\rightsquigarrow} (a^{i+1}_j, b^{i+1}_k) : \quad \text{normal transition for each chain}$$
$$(a^i_{j_0}, b^i_{k_0}) \overset{M'_i}{\rightsquigarrow} (a^{i+1}_{j_0}, b^{i+1}_{k_1}) : \quad \text{jump from chains } (a_{j_0}, b_{k_0}) \text{ to } (a_{j_0}, b_{k_1})$$

In order to simplify the notations, we often omit the chunk index, in order to show only the chains that are affected by the switch.

The main message chunk M_i and the secondary message chunk M'_i are determined when building the switch, and the main message defines the next state of all the chains. We note that the secondary message chunk M'_i should only be used when the state is $(a^i_{j_0}, b^i_{k_0})$. A simple example is depicted in Figure 2.

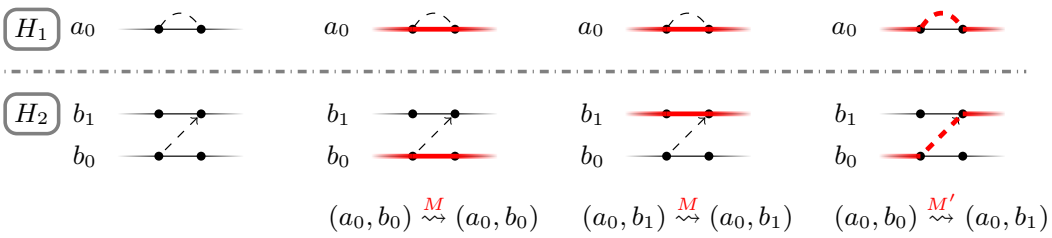


Fig. 2. A single switch: jump from (a_0, b_0) to (a_0, b_1) by using M' (dashed lines) instead of M (solid lines).

Alternatively, a switch can be designed to jump from (a_{j_0}, b_{k_0}) to (a_{j_1}, b_{k_0}). We defer the details of the construction to Section 3; it can be built with a complexity of $\tilde{O}(2^{n/2})$.

Interchange structure. By combining several simple switches, we can build an interchange structure with starting points IV_1 and IV_2 and ending points $\{A_j, j = 0 \dots 2^t - 1\}$ and $\{B_k, k = 0 \dots 2^t - 1\}$, so that we can select a message ending in any state (A_j, B_k). Figure 3 shows one possible way to build such a structure, and Figure 4 shows how to select a given message in the structure. An interchange structure with 2^t chains for each function requires about 2^{2t} switches. Since we can build a switch for a cost of $\tilde{O}(2^{n/2})$, the total structure is built with $\tilde{O}(2^{2t+n/2})$ operations.

Preimage search. Finally, we can use an interchange structure with ending points $\{A_j, j = 0 \dots 2^t - 1\}$ and $\{B_k, k = 0 \dots 2^t - 1\}$, to build a preimage

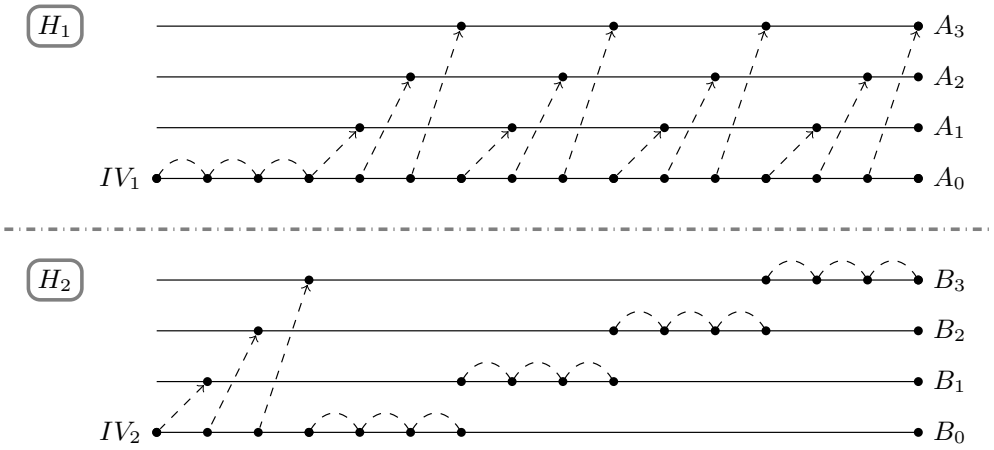


Fig. 3. Overview of an interchange structure

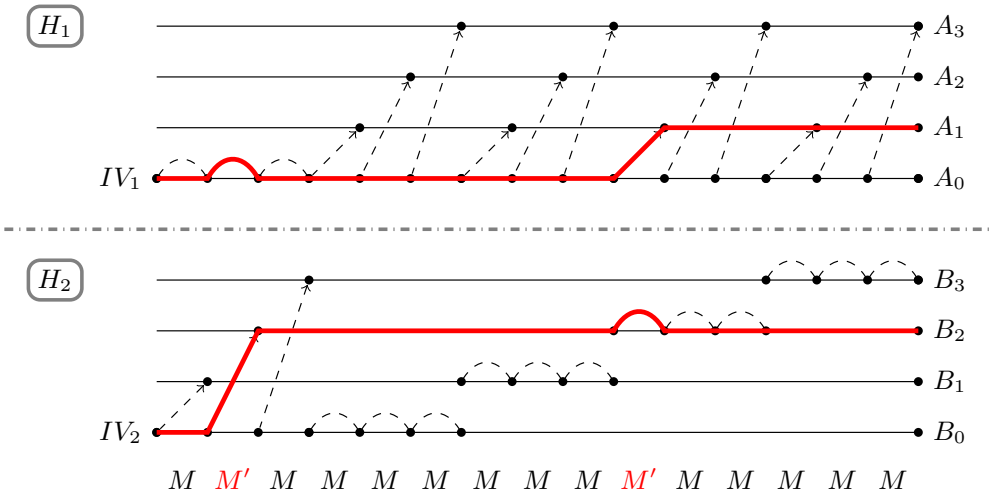


Fig. 4. Using of the interchange structure to reach output (A_1, B_2)

attack as follows. Let $\overline{H}$ denote the target value. We select a random message block m, and we compute two lists by evaluating the compression functions after the interchange structure: $\left\{A'_j = h_1(A_j, m), j = 0 \dots 2^t - 1\right\}$ and $\left\{B'_k = \overline{H} \oplus h_2(B_k, m), k = 0 \dots 2^t - 1\right\}$. We expect a match between the lists with probability 2^{2t-n}. After about 2^{n-2t} random choices of m, we get a match (j^*, k^*):

$$h_1(A_{j^*}, m) = \overline{H} \oplus h_2(B_{k^*}, m) \quad i.e. \quad h_1(A_{j^*}, m) \oplus h_2(B_{k^*}, m) = \overline{H}.$$

Therefore, we can construct a preimage of $\overline{H}$ by concatenating the message leading to (A_{j^*}, B_{k^*}) in the interchange structure, and the block m (we ignore the finalization function in this section).

The complexity of the preimage search is about 2^{n-t} evaluations of the compression function, using an interchange structure with 2^t end-points.

Complexity analysis. Building the interchange structures requires about $2^{2t+n/2}$ evaluations of the compression function, while the preimage search requires about 2^{n-t}. The optimal complexity is reached when both steps take the same time, *i.e.* $t = n/6$. This gives a complexity of $\tilde{O}(2^{5n/6})$.

3 The switch structure

We now explain how to build the switch structure at the core of our attack. This construction is strongly based on the multicollision technique of Joux [20].

Given states $a^i_{j_0}$, $b^i_{k_0}$ and $b^i_{k_1}$, we want to build message chunks M_i and M'_i in order to have the following transitions:

$$(a^i_{j_0}, b^i_{k_0}) \overset{M_i}{\rightsquigarrow} (a^{i+1}_{j_0}, b^{i+1}_{k_0}) \quad (a^i_{j_0}, b^i_{k_1}) \overset{M_i}{\rightsquigarrow} (a^{i+1}_{j_0}, b^{i+1}_{k_1}) \quad (a^i_{j_0}, b^i_{k_0}) \overset{M'_i}{\rightsquigarrow} (a^{i+1}_{j_0}, b^{i+1}_{k_1}).$$

The main message chunk M_i is used to define the next state of all the remaining chains, while the secondary message chunk M'_i will be used to jump from chains (a_{j_0}, b_{k_0}) to (a_{j_0}, b_{k_1}). We note that M'_i will only be used when the state is $(a^i_{j_0}, b^i_{k_0})$. In particular, M_i and M'_i must satisfy:

$$\begin{aligned} a^{i+1}_{j_0} &= h^*_1(a^i_{j_0}, M_i) = h^*_1(a^i_{j_0}, M'_i) \\ b^{i+1}_{k_1} &= h^*_2(b^i_{k_1}, M_i) = h^*_2(b^i_{k_0}, M'_i) \\ b^{i+1}_{k_0} &= h^*_2(b^i_{k_0}, M_i) \neq b^{i+1}_{k_1} \end{aligned}$$

We first build a multicollision for h^*_1, starting from state $a^i_{j_0}$, *i.e.* a large set $\mathcal{M}$ of $2^{n/2}$ messages that all reach the same state $a^{i+1}_{j_0}$ ($\forall M \in \mathcal{M}$, $h^*_1(a^i_{j_0}, M) = a^{i+1}_{j_0}$). As shown by Joux, this can be done efficiently by sequentially building $n/2$ collisions.

Next, we evaluate $h^*_2(b^i_{k_0}, M)$ and $h^*_2(b^i_{k_1}, M)$ for all the messages M in the set $\mathcal{M}$. With high probability there is match between the sets of values[8]. We denote the colliding messages as M_i and M'_i, so that we have $h^*_2(b^i_{k_0}, M'_i) = h^*_2(b^i_{k_1}, M_i)$.

Finally we compute the missing chains using the message m_i: $a^{i+1}_j = h^*_1(a^i_j, m_i)$, $b^{i+1}_k = h^*_2(b^i_k, m_i)$. With high probability all the chains reach distinct values; if this is not the case, we restart the construction with a new multicollision. The full algorithm is shown as Algorithm 1, and illustrated by Figure 5; it requires about $n/2 \cdot 2^{n/2}$ evaluations of the compression functions.

[8] If this is not the case, we build a new multicollision.

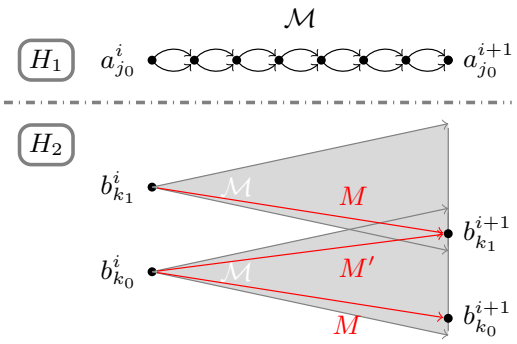


Fig. 5. Building a switch structure. First, M and M' are selected from $\mathcal{M}$ to generate a collision (defining the new b_{k_1}), then b_{k_0} is evaluated using M.

4 The interchange structure

Let us now describe the combination of switch structures into an interchange structure. The goal of this structure is to select the final value of the H_1 computation and the H_2 computation independently. More precisely, the structure defines two sets of final values A_j and B_k, and a set of messages $\boldsymbol{M}_{jk}$ such that:

$$(IV_1, IV_2) \overset{\boldsymbol{M}_{jk}}{\rightsquigarrow} (A_j, B_k).$$

In order to build this structure, we initialize the first chains with $a^0_0 = IV_1$, $b^0_0 = IV_2$, and set the other starting points randomly. Then, we use switches to jump for an already reachable pair (a_{j_0}, b_{k_0}) to a different pair (a_{j_0}, b_{k_1}) (or to (a_{j_1}, b_{k_0}), respectively). By using $2^{2t} - 1$ switches, we can make all pairs reachable. There are many way to combine the switches; a simple one can be described as follow:

1. first, build switches from (a_0, b_0) to each of the (a_0, b_k)'s;
2. then for each k, build a series of switches from (a_0, b_k) to all the (a_j, b_k)'s.

In order to reach the chains (a_j, b_k), one would activate the k-th switch in the first part to jump from (a_0, b_0) to (a_0, b_k), and then the j-th switch in the k-th series of the second part to jump from (a_0, b_k) to (a_j, b_k). This structure is shown in Figure 3 and a pseudo-code description is given by Algorithm 2, where the Interchange functions builds the structure, and the SelectMessage function extracts the message reaching (a_j, b_k).

The structure can be somewhat optimized using the fact that the extra chains have no prespecified initial values. We show how to take advantage of this in Appendix A, using multicollision structures in addition to the switch structures. However, this doesn't change significantly the complexity: we need $(2^t - 1)(2^t - 1)$ switches instead of $2^{2t} - 1$. In total, we need about $n/2 \cdot 2^{2t+n/2}$ evaluations of the compression functions to build a 2^t-interchange structure.

We believe that a 2^t-interchange structure based on switches will need at least $\Theta(2^{2t})$ switches, because every switch can only increase the number of reachable

Algorithm 1 Building a single switch

```
function SWITCH(h_1, h_2, a, b, b')
    x ← a
    M ← ∅
    for 0 ≤ i < n/2 do
        (m, m') ← COLLISION(h_1, x)
        M ← (M ‖ m) ∪ (M ‖ m')
        x ← h_1(x, m)
    end for
    H ← {}
    for M ∈ M do
        y ← h_2^*(b, M)
        H[y] ← M
    end for
    for M ∈ M do
        y ← h_2^*(b', M)
        if H[y] exists then
            return M, H[y]
        end if
    end for
end function

function COLLISION(h, x)
    H ← {}
    loop
        m ← $
        y ← h_1(x, m)
        if H[y] exists then
            return m, H[y]
        else
            H[y] ← m
        end if
    end loop
end function
```

pairs (a_j, b_k) by one. As shown in Appendix A some switches can be saved in the beginning but it seems that new ideas would be needed to reduce the total complexity below $\Theta(2^{2t+n/2})$.

5 Preimage Attack

Finally, we describe the full preimage attack. We first build an interchange structure with 2^t chains for each of H_1 and H_2. We denote the ending points as $\{A_j, j = 0 \dots 2^t - 1\}$ and $\{B_k, k = 0 \dots 2^t - 1\}$, and we know how to select a message $\boldsymbol{M}_{jk}$ to reach any state (A_j, B_k). When adding a message block m to one of the messages $\boldsymbol{M}_{jk}$ in the interchange structure, the output of the combiner

Algorithm 2 Building and using a T-interchange structure

```
function INTERCHANGE(h_1, h_2, IV_1, IV_2)
    a_0 ← IV_1, b_0 ← IV_2
    for 1 ≤ k < T do
        a_k ← $, b_k ← $
    end for
    for 1 ≤ j < T do
        (M, M') ← SWITCH(h_1, h_2, a_0, b_0, b_j)
        M ← M ‖ M; M' ← M' ‖ M'
        for 0 ≤ k < T do
            a_k ← h_1^*(a_k, M)
            b_k ← h_2^*(b_k, M)
        end for
    end for
    for 1 ≤ j < T do
        for 1 ≤ i < T do
            (M, M') ← SWITCH(h_2, h_1, b_j, a_0, a_i)
            M ← M ‖ M; M' ← M' ‖ M'
            for 0 ≤ k < T do
                a_k ← h_1^*(a_k, M)
                b_k ← h_2^*(b_k, M)
            end for
        end for
    end for
    return (M, M')
end function

function SELECTMESSAGE(M, M', j, k)
    μ ← M
    if k ≠ 0 then
        μ[k − 1] ← M'[k − 1]
    end if
    if j ≠ 0 then
        μ[(k + 1) · (T − 1) + j − 1] ← M'[(k + 1) · (T − 1) + j − 1]
    end if
    return μ
end function
```

can be written as:

$$H_1(\boldsymbol{M}_{jk} \parallel m) \oplus H_2(\boldsymbol{M}_{jk} \parallel m) = g_1(h_1(A_j, m), \ell + 1) \oplus g_2(h_2(B_k, m), \ell + 1),$$

where g_1 and g_2 are the finalization functions of H_1 and H_2, respectively, and ℓ is the length of the messages in the structure.

In order to reach a target value $\overline{H}$, we select a random block m, and we evaluate $\left\{A'_j = g_1(h_1(A_j, m), \ell + 1), j = 0 \dots 2^t - 1\right\}$ and $\left\{B'_k = \overline{H} \oplus g_2(h_2(B_k, m), \ell +\right.$

$1), k = 0 \dots 2^t - 1\big\}$. If there is a match (j^*, k^*) between the two lists, we have:

$$\begin{aligned} A'_{j^*} = B'_{k^*} &\Leftrightarrow g_1(h_1(A_{j^*}, m), \ell + 1) = \overline{H} \oplus g_2(h_2(B_{k^*}, m), \ell + 1) \\ &\Leftrightarrow H_1(\boldsymbol{M}_{jk} \parallel m) \oplus H_2(\boldsymbol{M}_{jk} \parallel m) = \overline{H}. \end{aligned}$$

For a random choice of m, we expect that a match exists with probability 2^{2t-n}, and testing it requires about 2^t operations[9]. We will have to repeat this procedure 2^{n-2t} times on average, therefore the total cost of the preimage search is about 2^{n-t} evaluations of h_1 and h_2.

As explained in the previous section, building a 2^t-interchange structure requires about $n/2 \cdot 2^{2t+n/2}$ operations. Using $t = n/6$ we balance the two steps of the attack, and reach the optimal complexity of about $n/2 \cdot 2^{5n/6}$ operations for the preimage attack.

5.1 Message length and memory complexity

The attack uses messages of length $n/2 \cdot 2^{2t}$, and the memory complexity of the attack[10] is also $n/2 \cdot 2^{2t}$. The optimal choice $t = n/6$ gives messages of length $n/2 \cdot 2^{n/3}$. The memory requirement is probably not an issue[11] for an attacker that can spend time $2^{5n/6}$, but the message length can be a problem with some hash functions that don't accept long inputs. For instance SHA-256 is only defined for message with less than 2^{64} bits (*i.e.* 2^{55} blocks).

In this case, one can apply the attack with a smaller value of t: this reduces the length of the messages, at the cost of more time spent in the preimage search step. For instance, we can mount a preimage attack against SHA-256 $\oplus$ BLAKE-256 with complexity 2^{232} using $t = 24$, while the optimal attack with $n = 256$ would cost only $2^{220.3}$. Similarly, our attack applied to SHA-512 $\oplus$ Whirlpool has a complexity of 2^{461}, rather than $2^{434.7}$.

6 Applications and Extensions

The attack works identically if the hash functions use the HAIFA mode rather than the plain Merkle-Damgård iteration. Also it can easily be extended to $H_1(M) \odot H_2(M)$ where $\odot$ denotes an easy to invert group operation (for instance, a modular addition rather than the exclusive or). The attack can also be extended to hash functions H_1 and/or H_2 using an internal check-sum, such as the GOST family of hash functions, using pairs of blocks with a constant sum.

[9] It takes $O(t \cdot 2^t)$ operations by sorting the lists, but only $2 \cdot 2^t$ using a hash table.
[10] We only need to store the messages $\boldsymbol{M}$ and $\boldsymbol{M}'$
[11] For instance, the attack is on the verge of practicality with $n = 64$; the time complexity is $2^{58.3}$ and the memory complexity is $2^{26.3}$.

6.1 Application to the sum of wide-pipe hash functions

The attack can also be used when the internal state size ℓ is larger than the output size n. The complexity of building a 2^t-interchange structure is related to ℓ as $\ell/2 \cdot 2^{2t+\ell/2}$. On the other hand, the complexity of the meet-in-the-middle preimage search is related to n as 2^{n-t}. The optimal complexity is $\ell/2 \cdot 2^{2n/3+\ell/6}$ by matching the two complexities with $t = n/3 - \ell/6$. Therefore our attack can be applied as long as $\ell + 6\log(\ell) \leq 2n$ holds. For instance, we can compute preimages of SHA-224 $\oplus$ BLAKE-224 with complexity roughly 2^{199}.

6.2 Application to Cryptophia's short combiner

Our attack can also be applied to Cryptophia's short combiner, as proposed by Mittelbach [32], and to the revised version of Mennink and Preneel [29]. This combiner computes the sum of two hash functions with some pre-processing of the message, to allow non-independent functions:

$$\begin{aligned} C(M) &= H_1\left(\tilde{m}_1^1 \parallel \ldots \parallel \tilde{m}_\ell^1\right) \oplus H_2\left(\tilde{m}_1^2 \parallel \ldots \parallel \tilde{m}_\ell^2\right) \\ \tilde{m}_j^1 &= H_1(0 \parallel l_1 \parallel m_j \oplus k_1) \oplus H_2(0 \parallel l_2 \parallel m_j \oplus k_2) \\ \tilde{m}_j^2 &= H_1(1 \parallel l_1 \parallel m_j \oplus k_1) \oplus H_2(1 \parallel l_2 \parallel m_j \oplus k_2) \end{aligned}$$

where k_1, k_2, l_1, l_2 is a randomly chosen key. The security proof in the ideal model shows that C is optimally preimage resistant if at least one of the hash functions is ideal.

However, if both H_1 and H_2 are narrow-pipe, we can apply our preimage attack with complexity $\tilde{O}(2^{5n/6})$. This does not violate the security proof because we need both functions to be narrow-pipe, hence not n-bit ideal[12]. From a practical point of view, though, it shows that in many cases (*e.g.* using SHA-512 and Whirlpool) the combiner is *weaker* than the initial functions.

6.3 Improvements using weaknesses of the hash functions

If H_1 or H_2 has known cryptographic weaknesses, more efficient attacks are possible. More precisely, if the compression function of one of the hash functions can be inverted[13] in time 2^t, then we can find a preimage of $H_1 \oplus H_2$ with complexity only $\tilde{O}(2^{(n+t)/2})$.

The attack is presented using the case, where (at least) one compression function is modeled as a weak compression function defined in [24], as an example. Without loss of the generality, we assume the compression function of H_2, $h_2(x, y) = z$, is such a weak compression function, which is a random oracle with two additional interfaces as below.

[12] A large multi-collisions can be built with a cost of roughly $2^{n/2}$ in a narrow-pipe function, but costs almost 2^n for an ideal hash function.

[13] finding an input chaining value from the output chaining value and the message

- *Backward interface.* On a query $(?, y, z)$, it returns either a value x uniformly chosen from all the values satisfying $h_2(x, y) = z$, or $\perp$ if no such x exists.

- *Bridging interface.* On a query $(x, ?, z)$, it return either a value y uniformly chosen from all the values satisfying $h_2(x, y) = z$, or $\perp$ if no such y exists.

Note that the inversion of compression function h_2 takes unit time and hence the attack against $H_1 \oplus H_2$ takes time $\tilde{O}(2^{n/2})$. The procedure is detailed as follows, which is also illustrated in Figure 6.

Let the target hash digest be denoted as $\overline{H}$. We firstly build an n-block long multicollision on H_1 following Joux's approach [20]. Let the final output be denoted as Z. It contains a set of up to 2^n messages that link IV_1 to Z on H_1. We split every n-block long multicolliding message into halves, and collect the first half $n/2$-block long messages as a set $\mathcal{M}_1$ and the second half as another set $\mathcal{M}_2$. Hence for any message $M_1 \in \mathcal{M}_1$ and any $M_2 \in \mathcal{M}_2$, the concatenated message $M_1 \parallel M_2$ links IV_1 to Z on H_1. Secondly, for the messages $M_2 \in \mathcal{M}_2$, we use the additional backward interface of h_2 to carry out backward computations from $\overline{H} \oplus Z$, and get values X such that $h_2^*(X, M_2) = \overline{H} \oplus Z$. We store (X, M_2) in a table T_X. On average T_X contains $2^{n/2}$ elements.[14] Finally, for each message $M_1 \in \mathcal{M}_1$, we iteratively compute h_2 forward from IV_2, and get an internal state Y. We match Y to the elements in T_X. A match implies that concatenated $M_1 \parallel M_2$ links IV_2 to $\overline{H} \oplus Z$, and in turn is a preimage of $H_1(M_1 \parallel M_2) \oplus H_2(M_1 \parallel M_2) = Z \oplus \overline{H} \oplus Z = \overline{H}$. The success probability of finding such a match is not negligible since there are $2^{n/2}$ X's and Y's. The complexity is around $n \cdot 2^{n/2}$ that is $\tilde{O}(2^{n/2})$ by ignoring the polynomial factors.

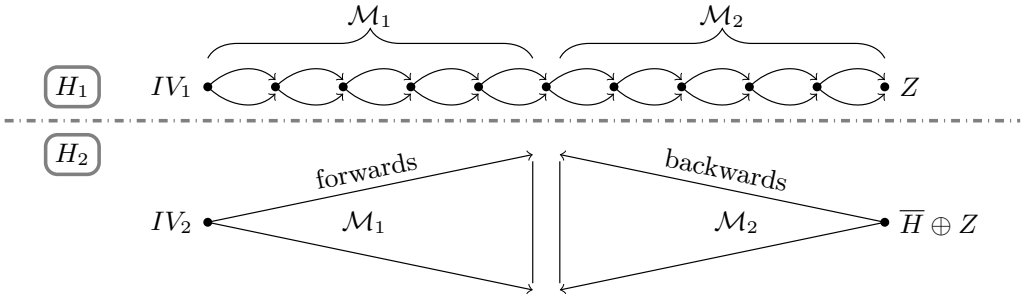


Fig. 6. Preimage attack on the XOR combiner with a weak H_2

Moreover, the above preimage attack can be extended to the concatenation combiner $H_1 \parallel H_2$ if both H_1 and H_2 are weak by a minor modification. This

[14] The backward interface may output $\perp$ for some message block. To compensate it, for the other message blocks, we make multiple queries, since they may have more than one preimages. On average, the backward interface should produce one preimage for each query.

shows that the proof of Hoch and Shamir [17] is tight for preimage resistance. Here we mainly highlight the modifications. Let the target hash digest be $\overline{H}_1 \| \overline{H}_2$, where $\overline{H}_1$ is from H_1 and $\overline{H}_2$ from H_2. After we build the multicollision on H_1 and let the output internal state be denoted as Z, we link Z to $\overline{H}_1$ by using the bridging interface of h_1 to receive a message m such that $h_1(Z, m) = \overline{H}_1$. This gives us a set of messages linking IV_1 to $\overline{H}_1$ on H_1. Also note that for the backward computations on H_2, the starting value should be $\overline{H}_2$. The rest of the attack procedure remains the same. Hence it is easy to get that the complexity is also $\tilde{O}(2^{n/2})$.

6.4 Extension to the sum of three or more hash functions

The attack can be extended to the sum of three or more hash functions. In order to attack the sum of k functions, two different strategies are possible: either we use a simpler structure that only gives two degrees of freedom, and fixes $k-2$ functions to a constant value, or we build an interchange structure to control all the k functions independently.

Controlling only two functions. The easiest way to extend the attack is to use a single chain in the $k-2$ extra hash functions. The procedure to build a switch is modified in order to use multicollisions for $k-1$ functions instead a simple multicollisions for one function; this costs $O(n^{k-1} \cdot 2^{n/2})$ using Joux's method [20].

As in the basic attack, we need $O(t^2)$ switches to generate a 2^t-interchange for two functions, and the preimage search costs $O(2^{n-t})$; the optimal complexity is therefore $O(n^{k-1} \cdot 2^{5n/6})$ with $t = n/6$.

Controlling all the functions. Alternatively, we can build a more complex interchange structure in order to control all the functions independently. When attacking three functions, we will use the switch structure to jump from chains $(a_{j_0}, b_{k_0}, c_{l_0})$ to $(a_{j_0}, b_{k_0}, c_{l_1})$ (or $(a_{j_0}, b_{k_1}, c_{l_0})$ or $(a_{j_1}, b_{k_0}, c_{l_0})$, respectively). We need $2^{3t} - 1$ switches in the interchange structure to reach all the 2^{3t} triplets of chains (a switch makes only one new triplet reachable). Each switch is built using a $2^{n/2}$-multicollision on two functions, which can be built for a cost of $O(n^2 \cdot 2^{n/2})$ following Joux's technique [20]. Therefore we can build a 2^t-interchange for a cost of $O(n^2 \cdot 2^{3t+n/2})$. More generally, for the sum of k hash functions, we can build an interchange structure for k functions for a cost of $O(n^{k-1} \cdot 2^{kt+n/2})$.

In the preimage search phase, we generate k lists of size 2^t, and we need to detect efficiently whether we can combine then to generate a zero sum. This problem can be solved using a algorithm similar to Wagner's generalized birthday algorithm [40]. If $k = 2^\kappa$, we find a solution with probability $O(2^{n-(\kappa+1)\cdot t})$ for a cost of $O(k \cdot 2^t)$. Therefore the preimage search costs $O(k \cdot 2^{n-\kappa t})$. With $k = 4$ (i.e. $\kappa = 2$), this yields a complexity of $O(n^3 \cdot 2^{5n/6})$. However, this approach is less efficient than the previous one for $k = 3$ and for $k > 4$.

To summarize, attacking the sum of k hash functions ($k \geq 2$) costs $O(n^{k-1} \cdot 2^{5n/6})$. Controlling chains independently in more than two hash function might be useful for further work, but it doesn't improve the preimage attack on the sum of k hash functions.

7 Conclusion and open discussions

In this work, we gave the first generic attack on the XOR combiner. Our result is rather surprising: the sum of two ideal narrow-pipe hash functions only has about $5n/6$ bits of security against preimage attacks. In particular, the *sum is easier to break* than the initial functions. Since most practical hash functions are narrow-pipe (*e.g.* SHA-1, SHA-256, SHA-512, Whirlpool, RIPEMD, GOST, BLAKE, Skein...), the XOR combiner will usually provide a weaker security than the component hash functions.

Moreover, we would like to discuss a few directions for future work.

On controlling multiple hash lanes. Since 2004, several generic attacks have been found against narrow-pipe hash functions, such as the multicollision attack[20], the long-message second preimage attack[22] and the herding attack[21]. There has been extensive work to extend these attacks to more complex constructions with several computation chains, such as the concatenation $H_1(M) \parallel H_2(M)$ and the cascade $H_2(H_1(M), M)$.

As in our present work, the essential difficulty in those attacks comes from the fact that several lanes of computation share the same input message, and hence their outputs are related. If an adversary considers a naïve set of messages, the set of outputs gives random pairs of n-bit values $\{(A_i, B_i) : i \in \mathcal{I}\}$: selecting a value for the first entry of the pair gives a single candidate for the second entry, and the adversary is essentially working with a $2n$-bit state. Previous works [33,16,2] have developed various message structures (mostly based on multicollision and diamond structures), in order to relax this relation. They mainly result in a set of messages $\mathcal{M}$ such that the corresponding outputs are in a more structured set $\{(A, B_i) : i \in \mathcal{I}\}$: the first entry is a constant value A, but several options B_i can be selected for the second entry. For any value (A, B_i), it is then possible to select a message in $\mathcal{M}$ so that the first lane reaches A, while the second lane reaches B_i. This allows to modify the value of the second lane without affecting the first lane.

Our result is quite stronger: with the interchange structure we have a set of message such that the corresponding outputs are a set $\{(A_i, B_j) : i \in \mathcal{I}, j \in \mathcal{J}\}$, where *both* lanes have several options that can be selected independently. We hope that our technique will have applications or lead to new technical development in related settings, e.g., the open problem of generic second preimage attacks (with long messages) on the concatenation hash or on the Zipper hash [24].

On extending to practical hash function. Several practical hash functions such as RIPEMD [10] and HAS-V [34] are based on a compression function with more than one independent lanes, which interacts with each other at the end of each compression function call. It is very interesting to investigate if our attack can be further modified to attack these hash functions in future. Again the obstacle comes from the relations between internal states of lanes. Particularly, the internal states of the lanes interact with each other, which makes the relation even tighter and in turn harder to attack.

Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments. Lei Wang is supported by the Singapore National Research Foundation Fellowship 2012 (NRF-NRFF2012-06).

References

1. Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.): Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations, Lecture Notes in Computer Science, vol. 5126. Springer (2008)
2. Andreeva, E., Bouillaguet, C., Dunkelman, O., Kelsey, J.: Herding, Second Preimage and Trojan Message Attacks beyond Merkle-Damgård. In: Jr., M.J.J., Rijmen, V., Safavi-Naini, R. (eds.) Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 5867, pp. 393–414. Springer (2009)
3. Biham, E., Dunkelman, O.: A Framework for Iterative Hash Functions - HAIFA. IACR Cryptology ePrint Archive, Report 2007/278 (2007)
4. Boneh, D., Boyen, X.: On the Impossibility of Efficiently Combining Collision Resistant Hash Functions. In: Dwork, C. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 4117, pp. 570–583. Springer (2006)
5. Brassard, G. (ed.): Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings, Lecture Notes in Computer Science, vol. 435. Springer (1990)
6. Cramer, R. (ed.): Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings, Lecture Notes in Computer Science, vol. 3494. Springer (2005)
7. Damgård, I.: A Design Principle for Hash Functions. In: Brassard [5], pp. 416–427
8. Dierks, T., Allen, C.: The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard) (Jan 1999), `http://www.ietf.org/rfc/rfc2246.txt`, obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176
9. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard) (Aug 2008), `http://www.ietf.org/rfc/rfc5246.txt`, updated by RFCs 5746, 5878, 6176

10. Dobbertin, H., Bosselaers, A., Preneel, B.: RIPEMD-160: A Strengthened Version of RIPEMD. In: Gollmann, D. (ed.) FSE. Lecture Notes in Computer Science, vol. 1039, pp. 71–82. Springer (1996)
11. Fischlin, M., Lehmann, A.: Multi-property Preserving Combiners for Hash Functions. In: Canetti, R. (ed.) TCC. Lecture Notes in Computer Science, vol. 4948, pp. 375–392. Springer (2008)
12. Fischlin, M., Lehmann, A., Pietrzak, K.: Robust Multi-property Combiners for Hash Functions Revisited. In: Aceto et al. [1], pp. 655–666
13. Fischlin, M., Lehmann, A., Pietrzak, K.: Robust Multi-Property Combiners for Hash Functions. J. Cryptology 27(3), 397–428 (2014)
14. Freier, A., Karlton, P., Kocher, P.: The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic) (Aug 2011), http://www.ietf.org/rfc/rfc6101.txt
15. Her, Y.S., Sakurai, K.: A Design of Cryptographic Hash Function Group with Variable Output-Length Based on SHA-1. Technical report of IEICE. ISEC 102(212), 69–76 (July 2002), http://ci.nii.ac.jp/naid/110003298501/en/
16. Hoch, J.J., Shamir, A.: Breaking the ICE - Finding Multicollisions in Iterated Concatenated and Expanded (ICE) Hash Functions. In: Robshaw [37], pp. 179–194
17. Hoch, J.J., Shamir, A.: On the Strength of the Concatenated Hash Combiner When All the Hash Functions Are Weak. In: Aceto et al. [1], pp. 616–630
18. Hong, D., Chang, D., Sung, J., Lee, S., Hong, S., Lee, J., Moon, D., Chee, S.: A New Dedicated 256-Bit Hash Function: FORK-256. In: Robshaw [37], pp. 195–209
19. Indesteege, S.: The lane hash function. Submission to NIST (2008), http://www.cosic.esat.kuleuven.be/publications/article-1181.pdf
20. Joux, A.: Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In: Franklin, M.K. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 3152, pp. 306–316. Springer (2004)
21. Kelsey, J., Kohno, T.: Herding Hash Functions and the Nostradamus Attack. In: Vaudenay, S. (ed.) EUROCRYPT. Lecture Notes in Computer Science, vol. 4004, pp. 183–200. Springer (2006)
22. Kelsey, J., Schneier, B.: Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work. In: Cramer [6], pp. 474–490
23. Lehmann, A.: On the Security of Hash Function Combiners. Ph.D. thesis, TU Darmstadt (2010)
24. Liskov, M.: Constructing an Ideal Hash Function from Weak Ideal Compression Functions. In: Biham, E., Youssef, A.M. (eds.) Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 4356, pp. 358–375. Springer (2006)
25. Mendel, F., Nad, T., Scherz, S., Schläffer, M.: Differential attacks on reduced RIPEMD-160. In: Gollmann, D., Freiling, F.C. (eds.) ISC 2012. Lecture Notes in Computer Science, vol. 7483, pp. 23–38. Springer (2012)
26. Mendel, F., Peyrin, T., Schläffer, M., Wang, L., Wu, S.: Improved Cryptanalysis of Reduced RIPEMD-160. In: Sako, K., Sarkar, P. (eds.) ASIACRYPT (2). Lecture Notes in Computer Science, vol. 8270, pp. 484–503. Springer (2013)
27. Mendel, F., Pramstaller, N., Rechberger, C., Rijmen, V.: On the collision resistance of RIPEMD-160. In: Katsikas, S.K., Lopez, J., Backes, M., Gritzalis, S., Preneel, B. (eds.) ISC 2006. Lecture Notes in Computer Science, vol. 4176, pp. 101–116. Springer (2006)
28. Mendel, F., Rechberger, C., Schläffer, M.: MD5 Is Weaker Than Weak: Attacks on Concatenated Combiners. In: Matsui, M. (ed.) ASIACRYPT. Lecture Notes in Computer Science, vol. 5912, pp. 144–161. Springer (2009)
29. Mennink, B., Preneel, B.: Breaking and Fixing Cryptophia's Short Combiner. In: Gritzalis, D., Kiayias, A. (eds.) CANS. Lecture Notes in Computer Science (2014)

30. Merkle, R.C.: One Way Hash Functions and DES. In: Brassard [5], pp. 428–446
31. Mittelbach, A.: Hash Combiners for Second Pre-image Resistance, Target Collision Resistance and Pre-image Resistance Have Long Output. In: Visconti, I., Prisco, R.D. (eds.) SCN. Lecture Notes in Computer Science, vol. 7485, pp. 522–539. Springer (2012)
32. Mittelbach, A.: Cryptophia's Short Combiner for Collision-Resistant Hash Functions. In: Jr., M.J.J., Locasto, M.E., Mohassel, P., Safavi-Naini, R. (eds.) ACNS. Lecture Notes in Computer Science, vol. 7954, pp. 136–153. Springer (2013)
33. Nandi, M., Stinson, D.R.: Multicollision Attacks on Some Generalized Sequential Hash Functions. IEEE Transactions on Information Theory 53(2), 759–767 (2007)
34. Park, N.K., Hwang, J.H., Lee, P.J.: HAS-V: A New Hash Function with Variable Output Length. In: Stinson, D.R., Tavares, S.E. (eds.) Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 2012, pp. 202–216. Springer (2000)
35. Pietrzak, K.: Non-trivial Black-Box Combiners for Collision-Resistant Hash-Functions Don't Exist. In: Naor, M. (ed.) EUROCRYPT. Lecture Notes in Computer Science, vol. 4515, pp. 23–33. Springer (2007)
36. Rjasko, M.: On Existence of Robust Combiners for Cryptographic Hash Functions. In: Vojtás, P. (ed.) ITAT. CEUR Workshop Proceedings, vol. 584, pp. 71–76. CEUR-WS.org (2009)
37. Robshaw, M.J.B. (ed.): Fast Software Encryption, 13th International Workshop, FSE 2006, Graz, Austria, March 15-17, 2006, Revised Selected Papers, Lecture Notes in Computer Science, vol. 4047. Springer (2006)
38. Sasaki, Y., Wang, L.: Distinguishers beyond Three Rounds of the RIPEMD-128/-160 Compression Functions. In: Bao, F., Samarati, P., Zhou, J. (eds.) ACNS. Lecture Notes in Computer Science, vol. 7341, pp. 275–292. Springer (2012)
39. Shoup, V. (ed.): Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings, Lecture Notes in Computer Science, vol. 3621. Springer (2005)
40. Wagner, D.: A Generalized Birthday Problem. In: Yung, M. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 2442, pp. 288–303. Springer (2002)
41. Wang, X., Yin, Y.L., Yu, H.: Finding Collisions in the Full SHA-1. In: Shoup [39], pp. 17–36
42. Wang, X., Yu, H.: How to Break MD5 and Other Hash Functions. In: Cramer [6], pp. 19–35

A Optimized Interchange Structure

We now describe an optimized attack using only $(2^t-1)(2^t-1)$ switches rather than $2^{2t}-1$. The attack also requires multicollision structures, as introduced by Joux[20].

We replace the first 2^t-1 switches with a 2^t-multicollision in H_1, and we use those messages to initialize all the b_k chains in H_2. We can also optimize the first series of switches in H_2 in the same way: we build a 2^t-multicollision in H_2 starting from b_0, and we use those messages to initialize the a_j chains in H_1. This is illustrated by Figure 7, and the detailed attack is given as Algorithm 3.

Algorithm 3 Optimized T-interchange structure

function INTERCHANGE(h_1, h_2, IV_1, IV_2, T)
  $a_0 \leftarrow IV_1$, $b_0 \leftarrow IV_2$
  $\mathcal{M}_0 \leftarrow$ MULTICOLLISION(h_1, a_0)
  for $0 \leq k < T$ **do**
    $b_k \leftarrow h_2^*(b_0, \mathcal{M}_0[k])$
  end for
  $a_0 \leftarrow h_1^*(a_0, \mathcal{M}_0[0])$
  $\mathcal{M}_1 \leftarrow$ MULTICOLLISION(h_2, b_0)
  for $1 \leq k < T$ **do**
    $a_k \leftarrow h_2^*(a_0, \mathcal{M}_1[k])$
  end for
  $a_0 \leftarrow h_2^*(a_0, \mathcal{M}_1[0])$
  $b_0 \leftarrow h_1^*(b_0, \mathcal{M}_1[0])$
  for $2 \leq j < T$ **do**
    for $1 \leq i < T$ **do**
      $(M, M') \leftarrow$ SWITCH(h_2, h_1, b_j, a_0, a_i)
      $\boldsymbol{M} \leftarrow \boldsymbol{M} \parallel M; \boldsymbol{M'} \leftarrow \boldsymbol{M'} \parallel M'$
      for $0 \leq k < T$ **do**
        $a_k \leftarrow h_1^*(a_k, M)$
        $b_k \leftarrow h_2^*(b_k, M)$
      end for
    end for
  end for
  return $(\mathcal{M}_0, \mathcal{M}_1, \boldsymbol{M}, \boldsymbol{M'})$
end function

function SELECTMESSAGE($\mathcal{M}_0, \mathcal{M}_1, \boldsymbol{M}, \boldsymbol{M'}, j, k$)
  if $j = 0$ **then**
    return $\mathcal{M}_0[k] \parallel \mathcal{M}_1[0] \parallel M$
  else if $k = 0$ **then**
    return $\mathcal{M}_0[0] \parallel \mathcal{M}_1[j] \parallel M$
  else
    $\mu \leftarrow M$
    $\mu[(k-1) \cdot (T-1) + j - 1] \leftarrow M'[(k-1) \cdot (T-1) + j - 1]$
    return $\mathcal{M}_0[k] \parallel \mathcal{M}_1[0] \parallel \mu$
  end if
end function

function MULTICOLLISION(h, x)
  $\mathcal{M} \leftarrow \{\}$
  for $0 \leq i < n/2$ **do**
    $(m, m') \leftarrow$ COLLISION(h, x)
    $x \leftarrow h(x, m)$
    $\mathcal{M} \leftarrow (\mathcal{M} \parallel m) \cup (\mathcal{M} \parallel m')$
  end for
  return $\mathcal{M}$
end function

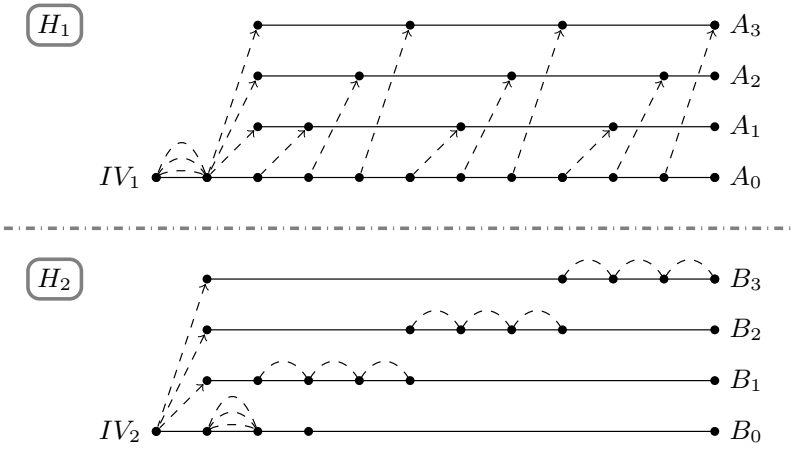


Fig. 7. Optimized interchange structure

Generic Attacks on Hash Combiners

Zhenzhen Bao[1,2], Itai Dinur[3], Jian Guo[1], Gaëtan Leurent[4], and Lei Wang[5,6]

[1] Division of Mathematical Sciences, School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore
baozhenzhen10@gmail.com, guojian@ntu.edu.sg
[2] Strategic Centre for Research in Privacy-Preserving Technologies and Systems, Nanyang Technological University, Singapore
[3] Department of Computer Science, Ben-Gurion University, Israel
dinuri@cs.bgu.ac.il
[4] Inria, France
gaetan.leurent@inria.fr
[5] Shanghai Jiao Tong University, Shanghai, China
[6] Westone Cryptologic Research Center, Beijing, China
wanglei_hb@sjtu.edu.cn

Abstract. Hash combiners are a practical way to make cryptographic hash functions more tolerant to future attacks and compatible with existing infrastructure. A combiner combines two or more hash functions in a way that is hopefully more secure than each of the underlying hash functions, or at least remains secure as long as one of them is secure. Two classical hash combiners are the exclusive-or (XOR) combiner $\mathcal{H}_1(M) \oplus \mathcal{H}_2(M)$ and the concatenation combiner $\mathcal{H}_1(M) \| \mathcal{H}_2(M)$. Both of them process the same message using the two underlying hash functions in parallel. Apart from parallel combiners, there are also cascade constructions sequentially calling the underlying hash functions to process the message repeatedly, such as Hash-Twice $\mathcal{H}_2(\mathcal{H}_1(IV, M), M)$ and the Zipper hash $\mathcal{H}_2(\mathcal{H}_1(IV, M), \overleftarrow{M})$, where $\overleftarrow{M}$ is the reverse of the message M.

In this work, we study the security of these hash combiners by devising the best-known generic attacks. The results show that the security of most of the combiners is not as high as commonly believed. We summarize our attacks and their computational complexities (ignoring the polynomial factors) as follows:

1. Several generic preimage attacks on the XOR combiner:
   - A first attack with a best-case complexity of $2^{5n/6}$ obtained for messages of length $2^{n/3}$. It relies on a novel technical tool named Interchange Structure. It is applicable for combiners whose underlying hash functions follow the Merkle-Damgård construction or the HAIFA framework.
   - A second attack with a best-case complexity of $2^{2n/3}$ obtained for messages of length $2^{n/2}$. It exploits properties of functional graphs of random mappings. It achieves a significant improvement over

[§] This paper is a combination and extension of three conference papers [LW15,Din16,BWGG17].

the first attack but is only applicable when the underlying hash functions use the Merkle-Damgård construction.
- An improvement upon the second attack with a best-case complexity of $2^{5n/8}$ obtained for messages of length $2^{5n/8}$. It further exploits properties of functional graphs of random mappings and uses longer messages.

These attacks show a rather surprising result: regarding preimage resistance, the sum of two n-bit narrow-pipe hash functions following the considered constructions can never provide n-bit security.

2. A generic second-preimage attack on the concatenation combiner of two Merkle-Damgård hash functions. This attack finds second preimages faster than 2^n for challenges longer than $2^{2n/7}$ and has a best-case complexity of $2^{3n/4}$ obtained for challenges of length $2^{3n/4}$. It also exploits properties of functional graphs of random mappings.
3. The first generic second-preimage attack on the Zipper hash with underlying hash functions following the Merkle-Damgård construction. The best-case complexity is $2^{3n/5}$, obtained for challenge messages of length $2^{2n/5}$.
4. An improved generic second-preimage attack on Hash-Twice with underlying hash functions following the Merkle-Damgård construction. The best-case complexity is $2^{13n/22}$, obtained for challenge messages of length $2^{13n/22}$.

The last three attacks show that regarding second-preimage resistance, the concatenation and cascade of two n-bit narrow-pipe Merkle-Damgård hash functions do not provide much more security than that can be provided by a single n-bit hash function.

Our main technical contributions include the following:

1. The interchange structure, which enables simultaneously controlling the behaviours of two hash computations sharing the same input.
2. The simultaneous expandable message, which is a set of messages of length covering a whole appropriate range and being multi-collision for both of the underlying hash functions.
3. New ways to exploit the properties of functional graphs of random mappings generated by fixing the message block input to the underlying compression functions.

Keywords: Hash function, Generic attack, Hash combiner, XOR combiner, Concatenation combiner, Zipper hash, Hash-Twice, (Second) Preimage attack

1 Introduction

A cryptographic hash function $\mathcal{H} : \{0,1\}^* \to \{0,1\}^n$ maps arbitrarily long messages to n-bit digests. It is a fundamental primitive in modern cryptography and is among the main building blocks of many widely utilized cryptographic protocols and cryptosystems. There are three *basic* security requirements on a hash function $\mathcal{H}$:

- **Collision resistance**: It should be computationally infeasible to find a pair of different messages M and M' such that $\mathcal{H}(M) = \mathcal{H}(M')$.
- **Preimage resistance**: Given an arbitrary n-bit value V, it should be computationally infeasible to find any message M such that $\mathcal{H}(M) = V$.
- **Second-preimage resistance**: Given a challenge message M, it should be computationally infeasible to find any different message M' such that $\mathcal{H}(M) = \mathcal{H}(M')$.

As the birthday and brute-force attack requires $2^{n/2}$ and 2^n computations, respectively, to find a collision and a (second) preimage, a secure hash function is expected to provide the same level of resistance.

Unfortunately, widely deployed standards (such as MD5 and SHA-1) fail to provide the expected resistance because of cryptographic weaknesses [WYY05,WY05]. Moreover, Kelsey and Schneier have demonstrated a generic second-preimage attack against all hash functions based on the classical Merkle-Damgård construction (such as MD5, SHA-1, and SHA-2) when the challenge message is long [KS05]. As a result, countermeasures have been proposed in order to build more tolerant hash functions, and to protect oneself against future attacks, while keeping the same interface for compatibility. A practical way is to combine the outputs of two (or more) independent hash functions to provide better security in case one or even both hash functions are weak. In particular, this reasoning was used by the designers of SSL [FKK11] and TLS [DA99b], who combined MD5 and SHA-1 in various ways. More precisely, the Key Derivation Function of TLS v1.0/v1.1 uses a sum of HMAC-MD5 and HMAC-SHA-1.[7] The designers explain [DA99b], "In order to make the PRF as secure as possible, it uses two hash algorithms in a way which should guarantee its security if either algorithm remains secure." Formally, we call the resulting construction a *hash function combiner* (or combiner for short). The goal of a hash combiner is to achieve *security amplification*, *i.e.*, the hash combiner has higher security than its underlying hash functions, or to achieve *security robustness*, *i.e.*, the hash combiner is secure as long as (at least) one of its underlying hash functions is secure.

There are two classical hash combiners, the concatenation combiner and the exclusive-or (XOR) combiner. Both of them process the same message using two (independent) hash functions $\mathcal{H}_1$ and $\mathcal{H}_2$ in parallel. Then, the former concatenates their outputs, $\mathcal{H}_1(M) \parallel \mathcal{H}_2(M)$, and the latter XORs them, $\mathcal{H}_1(M) \oplus \mathcal{H}_2(M)$. More generally[8], cryptographers have also studied cascade constructions of two (or more) hash functions, that is, to compute $\mathcal{H}_1$ and $\mathcal{H}_2$ in sequential order. Examples are Hash-Twice, $\mathcal{HT}(M) \triangleq \mathcal{H}_2(\mathcal{H}_1(IV, M), M)$[9], and the Zipper hash [Lis06], $\mathcal{H}_2(\mathcal{H}_1(IV, M), \overleftarrow{M})$, where $\overleftarrow{M}$ is the message with the same blocks as M but in reversed order. Such kinds of cascade constructions

[7] We note that this MD5/SHA-1 combiner has been replaced by primitives based on single hash function (*e.g.*, SHA-256) since TLS v1.2 [DR08].

[8] Here, we generalize the syntax of hash functions to also regard the initial value IV as an input parameter.

[9] The original specification of Hash-Twice is $\mathcal{HT}(M) \triangleq \mathcal{H}(\mathcal{H}(IV, M), M)$, which processes the same message twice using a single hash function as shown in [ABDK09].

are not strictly black-box hash combiners compared with the XOR combiner and the concatenated combiner. That is because the initial vector of the subsequent hash functions are not fixed as in their specifications. Instead, for such constructions, to get black-box access to the underlying hash functions, the initial vector of these hash functions is required to be an input parameter. However, apart from this point, other parts of the hash functions are all accessed as black boxes by the constructions. Considering that such constructions can be classified into the method of designing hash functions using multiple existing hash functions, and for the sake of conciseness, we regard these cascade constructions of hash functions as hash combiners in this paper. In the sequel, for these combiners, we call the computation of the first (resp. the second) hash function the first (resp. the second) computation pass (or simply, the *first pass*, resp. the *second pass*).

In this paper, we study the security of these hash combiners. We focus on combiners of *iterated* hash functions. Iterated hash functions commonly first pad and split the message M into message blocks of fixed length (*e.g.*, b-bit), *i.e.*, $M = m_1 \parallel m_2 \parallel \ldots \parallel m_L$. They then process message blocks by iteratively applying a series of compression function h_i. Those compression functions update an internal state, initialized with a public value IV, using the previous state value and the current message block, *i.e.*, $x_i = h_i(x_{i-1}, m_i)$. Finally, the internal state is updated by a finalization function which can be either the compression function or another independent function, and the output of the finalization function is outputted as the hash digest. For simplicity of description, we assume that the finalization function is the same as the compression function in the rest of the paper, but we stress that our attacks also work in a straightforward way with an independent finalization function. In this work, we mainly focus on hash functions whose internal state size is equal to its output size, which are known as "narrow-pipe" designs. We will discuss the applicability of our proposed attacks on "wide-pipe" designs whose internal state size is (not much) larger than its output size at the end of the paper. In particular, we consider hash functions following the classical Merkle-Damgård construction [Dam89,Mer89] and the more general HAIFA framework [BD07], for which we naturally take the length padding mentioned next as the message padding scheme. The Merkle-Damgård construction (MD) applies the same compression function h in all iterations (see Fig. 1) and adds a padding of the message length to the final message block (known as length padding, or Merkle-Damgård strengthening). The HAIFA framework is similar to the MD construction but uses extra inputs to the compression function with the number of message bits hashed so far and a salt value; this is equivalent to using a different compression function for each block. The primary goal of that construction is to thwart some narrow-pipe attacks (*e.g.*, Kelsey and Schneier's long message second-preimage attack on MD hash [KS05]). HAIFA framework is formally defined as follows:[10]

$$x_0 = IV \qquad x_i = h(x_{i-1}, m_i, \#bits, salt) \qquad \mathcal{H}(M) = h(x_{L-1}, m_L).$$

[10] For simplicity of description, we omit the computation of the initial value $IV_n = h(IV, n, 0, 0)$, which is used to support variable hash size in the specification of HAIFA in [BD07]. This does not influence the attacks.

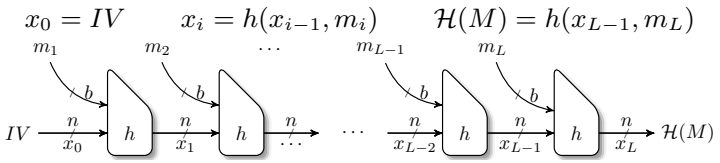


Fig. 1: Narrow-pipe Merkle-Damgård hash function

1.1 Related Works

Combiners have been studied in several settings, including generic attacks and security proof. For generic attacks, the compression functions are modelled as random functions to devise attacks that do not use any weakness of the compression function. Thus, they provide upper bounds on the security of the combiners.

Security proof, which is more theoretical work, focuses on the notion of *robustness* and *security amplification*. A robust combiner is secure regarding property α as long as one of the underlying hash functions is secure regarding α. Lines of research for those security notions include the study of advanced combiners in [Her05,CRS+07,FL07,FL08,FLP08,Her09,Leh10,FLP14]. A series of studies on the minimum output length of robust combiners have shown that robust combiners for collision resistance and preimage resistance cannot have an output length significantly shorter than the sum of the output length of the underlying hash functions [BB06,Pie07,Pie08,Rja09] (more recent works include [MP14,Mit12]).

There are also some works that assume that the compression function is a *weak* random oracle (*i.e.*, the attacker is given additional interfaces to receive random preimages of the compression functions) and prove that some constructions are still secure in this model [Lis06,HS08] or provide efficient attacks on the combiners by exploiting weakness of the underlying hash functions [CJ15,JN15].

Analysis of the concatenation combiner. The concatenation combiner $\mathcal{H}_1(M) \,\|\, \mathcal{H}_2(M)$ (see Fig. 2) is probably the most well-known and most studied hash combiner. This combiner was described already in 1993 [Pre93].

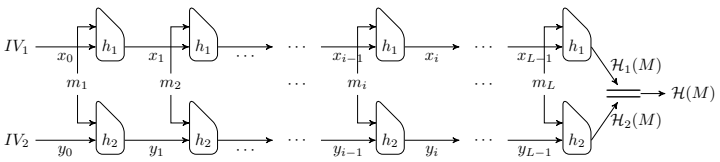


Fig. 2: The concatenation combiner

Generic attacks. In 2004, Joux [Jou04] described surprising attacks on the concatenation of two narrow-pipe iterated hash functions using multi-collisions: while the output size is $2n$ bits, the concatenation combiner merely provides at most $n/2$-bit security for collision resistance and n-bit security for preimage resistance[11] (see Sect. 2.1 for a description). In particular, the concatenation combiner is not security-amplifying (it does not increase collision and preimage resistance).

Following the results of Joux (which showed that the concatenation combiner does not increase collision and preimage resistance) and the later results of Kelsey and Schneier (which showed that the second-preimage resistance of the MD construction is less than 2^n), a natural question is whether there exists a second-preimage attack on the concatenation combiner of MD hash functions that is faster than 2^n. Interestingly, the problem of devising such an attack remained open for a long time despite being explicitly mentioned in several papers including [DP07]. In fact, although the works of [Jou04,KS05] have attracted a significant amount of follow-up research on countermeasures against second-preimage attacks (such as Hash-Twice or dithered hash) and attacks that break them [ABD+16,ABDK09,ABF+08], there has been no progress concerning second-preimage attacks on the basic concatenation combiner. In this paper, we try to provide an answer to this question by devising the first second-preimage attack on the concatenation combiner of MD hash functions, which is faster than 2^n.

Security proof. From the theoretical side, the concatenation combiner is robust for collision resistance, *e.g.*, a collision $\mathcal{H}_1(M) \parallel \mathcal{H}_2(M) = \mathcal{H}_1(M') \parallel \mathcal{H}_2(M')$ implies $\mathcal{H}_1(M) = \mathcal{H}_1(M')$ and $\mathcal{H}_2(M) = \mathcal{H}_2(M')$.

Hoch and Shamir [HS08] evaluated the security of the concatenation combiner with two weak hash functions. More precisely, the two hash functions are narrow-pipe MD, and the compression functions are modelled as weak random oracles (as defined by Liskov [Lis06]), *i.e.*, the adversary is given additional interfaces to receive (random) preimages of the compression functions. They have proven that in this model, the concatenation combiner is still indifferentiable from a random oracle with $n/2$-bit security, implying (at least) the same security bound for collision resistance and preimage resistance. The bound is matched by Joux's attack for collisions, but there is a gap with Joux's attack for preimages, with complexity 2^n, which might be interesting to investigate further.

Analysis of dedicated instantiations. Mendel *et al.* analysed some dedicated instantiations of the concatenation combiner [MRS09], in particular using the hash function MD5. We omit the details and refer interested readers to [MRS09].

Analysis of the XOR combiner. The XOR combiner $\mathcal{H}_1(M) \oplus \mathcal{H}_2(M)$ (see Fig. 3) has received less analysis.

[11] The attacks essentially only require one of the functions to be iterated.

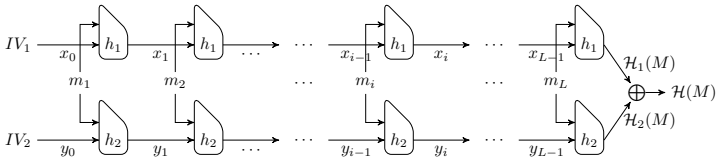


Fig. 3: The XOR combiner

Generic attacks. To the best of our knowledge, no preimage attacks have been shown against the XOR combiner. Therefore, the preimage security of the XOR combiner against generic attacks is still an open problem and will be one of the main topics of our work.

Security proof. Theoretically, the XOR combiner is robust concerning PRF (Pseudo-Random Function) and MAC (Message Authentication Code) in the black-box reduction model [Leh10]. Since the XOR combiner is length-preserving, from the conclusions regarding the minimum output length of robust combiners, it is not robust for collision resistance and preimage resistance. However, the work of Hoch and Shamir [HS08] actually proves the security of the XOR combiner as an intermediate result: it is also indifferentiable from a random oracle up to $2^{n/2}$ queries in the weak random oracle model. In particular, this proves there are no generic attacks with complexity less than $2^{n/2}$. For collision resistance, the bound is tight, since it is matched with the generic birthday attack bound. On the other hand, for preimage resistance, there exists a gap between the $n/2$-bit proven bound and the n-bit expected ideal security bound. Note that the non-robustness result regarding preimage security does not imply that the XOR of two concrete hash functions is weak, and the simplicity and short output of this construction still make it quite attractive.

Analysis of Hash-Twice. Hash-Twice is a folklore hash construction that hashes a (padded) message twice, with the output of the first hash value as the value of the initialization vector of the second hash. In its original definition [ABDK09], the two underlying hash functions are identical, *i.e.*, $\mathcal{HT}(M) \triangleq \mathcal{H}(\mathcal{H}(IV, M), M)$; here, we consider a generalized version, where the underlying hash functions are independent, *i.e.*, $\mathcal{HT}(M) \triangleq \mathcal{H}_2(\mathcal{H}_1(IV, M), M)$ (see Fig. 4).

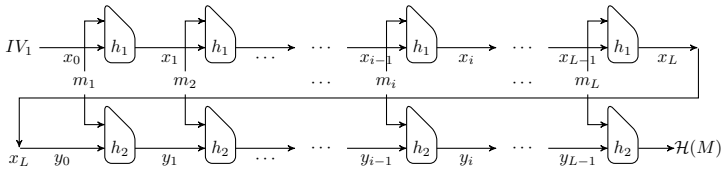


Fig. 4: The Hash-Twice

Generic attacks. Towards the three basic security requirements, a second-preimage attack on Hash-Twice ($\mathcal{HT}(M) \triangleq \mathcal{H}(\mathcal{H}(IV, M), M)$) has been published by An-

dreeva *et al.* in [ABDK09]. The attack is based on a herding attack, which exploits the diamond structure originally used in the herding attack on a single hash function [KK06] (see Sect. 2.3 for an introduction). The complexity of the attack is approximately $2^{(n+t)/2} + 2^{n-\ell} + 2^{n-t}$, where 2^t is the width of the diamond structure, and 2^ℓ is the length of the challenge.

Security proof. To the best of our knowledge, there is no published formal proof regarding the security of Hash-Twice. However, we can claim that they are at least as secure as the original functions: a generic collision attack requires at least $2^{n/2}$ (because we need a collision in one of the compression functions); a preimage attack requires at least 2^n (because we need a preimage for the finalization function); a second-preimage requires at least $2^{n/2}$ (because it implies a collision).

Analysis of the Zipper hash. The Zipper hash has been proposed with the goal of constructing an ideal hash function from weak ideal compression functions (by "weak ideal", it means that the compression function is vulnerable to strong forms of attack but is otherwise random). Similar to Hash-Twice, it cascades two independent hash functions evaluating the same (padded) message. The difference is that the second hash processes the message blocks in reverse order, *i.e.*, $\mathcal{ZH} \triangleq \mathcal{H}_2(\mathcal{H}_1(IV_1, M), \overleftarrow{M})$ (see Fig. 5). Note that the messages are first padded by a padding scheme and split into message blocks, and then they are processed in forward and reverse order sequentially. Thus, the padded message block m_L is processed at the end of the first hash computation and at the beginning of the second hash computation, *i.e.*, in the middle of the whole processing procedure. The padding scheme of Zipper was specified to be any injective function of the message [Lis06]. In this paper, and as for all other combiners, we take the length padding of the MD construction as the padding scheme.

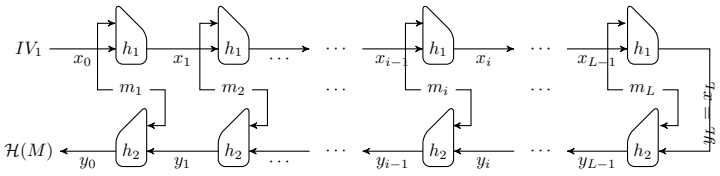


Fig. 5: The Zipper hash

Generic attacks. To the best our knowledge, no generic attacks on the Zipper hash regarding the three basic security notions have been shown. However, there are a number of works that consider other security notions, such as multi-collision, herding attack or attacks assuming weak compression functions. Examples include [NS07,HS06,ABDK09,CJ15,JN15], some of which also consider the corresponding security of Hash-Twice.

Security proof. Zipper hash is proved ideal in the sense that the overall hash function is indistinguishable from a random oracle (up to the birthday bound) when instantiated with weak ideal compression functions. More precisely, its provable security is $2^{\min(b,n)/2}$ for collision resistance and $2^{\min(b,n)}$ for (second-) preimage resistance, where b is the size of the message block and n is the size of the internal state and the hash value (considering "narrow-pipe" design).

1.2 Our Results

In this work, we study the upper bound of the security of these hash combiners by devising the best-known generic preimage attacks and second-preimage attacks with a long challenge. We do not assume any weakness of the compression functions, *i.e.*, they are accessed as black boxes in our attacks. The compression functions are chosen uniformly at random from all $n+b$ to n-bit functions, which implies that our analysis applies to most compression functions. Table 1 summarizes the updated security status of various hash combiners after integrating our new results. It shows that the security of most combiners is not as high as commonly believed. Regarding certain basic security requirements, these combiners of two (or even more) n-bit hash functions fail to provide more (or even the same) security than that provided by a single n-bit ideal hash function.

Next, we briefly introduce our main attacks on combiners of two narrow-pipe hash functions and their computational complexities (ignoring the polynomial factors). In addition to the attacks summarized next, we also present improved but more complex attacks and apply our techniques to attack combiners of more than two hash functions.

Preimage attacks on the XOR combiner. We present several generic attacks:

- *A first attack with a best-case complexity of* $2^{5n/6}$ *computations obtained for messages of length* $2^{n/3}$. This attack involves a meet-in-the-middle procedure enabled by building a novel technical tool named *interchange structure*. This structure consists of a sequence of basic building modules named *switches* among different hash computation lanes such that it breaks the pairwise dependency between the internal states of these hash computations on the same message. This attack is applicable for the XOR combiner with underlying hash functions following a wide range of iterated constructions (*e.g.*, the classical MD construction and the more general HAIFA framework) and for the Cryptophia's short combiner [Mit13,MP14].
- *A second attack with a best-case complexity of* $2^{2n/3}$ *computations obtained for messages of length* $2^{n/2}$. This attack also involves a meet-in-the-middle procedure, but instead of using the interchange structure, it exploits properties of functional graphs of random mappings. The random mappings are generated by fixing the message block input to the underlying compression functions. The functional graphs of these random mappings are formed by

successive iteration of the mappings, whose nodes are all possible values of input/output and whose edges are all from preimages to images. We exploited special nodes that are images of a large number of iterations of the mappings. We named them *deep iterates* because they are located at deep strata in the functional graphs. By exploiting such deep iterates, the variability of the number of iterations provides extra freedom to find a linking message fragment mapping a pair of starting states to a predefined pair of states. When using this freedom, to overcome the hurdle set by the length padding, we construct a structure named *simultaneous expandable message*, which is a set of messages of length covering a whole appropriate range and being multi-collisions for both of the underlying hash functions. This attack achieves a trade-off of $2^n \cdot L^{-2/3}$ between the maximal allowed message length L and the time complexity of attack. This improves the trade-off of $2^n \cdot L^{-1/2}$, obtained by the first interchange-structure-based attack. On the other hand, it only applies when both underlying hash functions combined use the MD construction.

– *An improvement upon the second attack with a best-case complexity of* $2^{5n/8}$ *obtained for messages of length* $2^{5n/8}$. In this attack, we exploit more special nodes in functional graphs of random mappings, which are called cyclic nodes, and we utilize a technique named *multi-cycles*. Linking pairs of states through cyclic nodes and allowing to loop around the cycles makes the attack more efficient. The complexity improvement of the resulting preimage attack is $2^{n/24}$. We point out that this attack has the limitation that the length of the message must be at least $2^{n/2}$ blocks. Therefore, its practical impact is limited and its main significance is theoretical. This attack shows that the security level regarding preimage resistance of the XOR combiner is quite close to the provable security level $2^{n/2}$, which is also the level of collision resistance.

These attacks on the XOR combiner show a rather surprising result – regarding preimage resistance, the sum of two n-bit iterated hash functions can never provide n-bit security. In general, the sum is *weaker* than each of the two hash functions.

Second-preimage attack on the concatenation combiner. We describe the first generic second-preimage attack faster than 2^n on the concatenation combiner of two MD hash functions. The general framework follows that of the long message second-preimage attack on a single MD hash [KS05]. It is faster than 2^n computations by overcoming two main challenges. The first is to overcome the length padding. We solve this by using the *simultaneous expandable message*. The second is to speed up the connection from a crafted message to the challenge message on chaining states. In the case of hash combiners, one must connect to the challenge message on *a pair of* n-bit states, while in the second-preimage attack on a single MD hash, one needs only to connect on a single n-bit state. Thus, the attempt is essentially to reach a $2n$-bit state (in a set of L states, where L is the message length in blocks, and thus $L < 2^n$) faster than

2^n computations. We solve this by exploiting again deep iterates in functional graphs. In this attack, we choose a pair of deep iterates as target chaining states, such that the connection from our crafted message to the challenge message is more efficient. Indeed, this attack is closely related with our deep-iterates-based preimage attack on the XOR combiners.

We obtain a trade-off between the complexity of the attack and the length of the challenge message. This second-preimage attack is faster than 2^n for input messages of length at least[12] $2^{2n/7}$. The best-case complexity[13] is $2^{3n/4}$, obtained for (very) long challenges of length $2^{3n/4}$. Again, due to these constraints, the practical impact of our second-preimage attack is limited and its main significance is theoretical. Namely, it shows that regarding second-preimage resistance, the concatenation of two n-bit MD hash functions is not as strong as a single n-bit ideal hash function.

Second-preimage attack on the Zipper hash. We show the first generic second-preimage attack on the Zipper hash. This attack combines multiple tools, including Joux's multi-collision, the simultaneous expandable message, deep iterates and multi-cycles in functional graph of random mappings. The general framework is similar to that of above ones on combiners of MD hashes. However, some special specifications of the Zipper hash allow the attacker to choose an optimal configuration on message length for the attack, and to launch a more efficient meet-in-the-middle connecting procedure in the attack. The best-case complexity of this attack is $2^{3n/5}$, obtained for challenge message of length $2^{2n/5}$. This result shows that combination of two MD hash functions using a Zipper can be vulnerable to second-preimage attack with long challenges.

Second-preimage attack on Hash-Twice. We give an improved second-preimage attack on Hash-Twice. This attack also combines multiple tools including Joux's multi-collision, the diamond structure, the interchange structure, the simultaneous expandable message, deep iterates and multi-cycles in functional graphs. Like all our previous functional-graph-based attacks, it improves a previous attack from [ABDK09] because of the efficiency brought by exploiting the special nodes in the functional graphs. The best-case complexity of this attack is $2^{13n/22}$, obtained for challenge message of length $2^{13n/22}$. This attack shows that regarding second-preimage resistance, hashing a message twice using two Merkle-Damgård hash functions does not provide much more security compared with hashing the message only once.

Finally, we highlight the technical interests of this paper. We believe that the tools introduced in this paper – the interchange structure, the simultaneous expandable message, deep iterates and multi-cycles in random functional

[12] For example, for $n = 160$ and message block of length 512 bits (as in SHA-1), the attack is faster than 2^{160} for messages containing at least 2^{46} blocks, or 2^{52} bytes.

[13] The complexity formulas do not take into account (small) constant factors, which are generally ignored throughout this paper.

graphs – are important technical advances and will hopefully have further applications or lead to new technical developments in related settings. Particularly, we point out that we can use the interchange structure in order to optimize the complexity of functional-graph-based attacks on the XOR combiner, concatenation combiner, and Hash-Twice. Although this does not lead to a very big improvement, it further demonstrates the wide applicability of this structure in cryptanalysis of hash function combiners.

$\mathcal{H}$	Collision		Preimage		2nd Preimage	(challenge length)
Ideal	$2^{0.5n}$	Birthday	2^n	Brute-force	2^n	Brute-force
HAIFA	-	-	-	-	-	-
MD	-	-	-	-	$2^{0.5n}$	[KS05] DS ($2^{0.5n}$)
Provable	$2^{0.5n}$		2^n		$2^{0.5n}$	

$\mathcal{H}_1 \oplus \mathcal{H}_2$	Collision		Preimage		2nd Preimage	
Ideal	$2^{0.5n}$	Birthday	2^n	Brute-force	2^n	Brute-force
HAIFA	-	-	$2^{0.833n}$	[Sect. 3.1] IS		
MD	-	-	$2^{0.833n}$	[Sect. 3.1] IS		
	-	-	$2^{0.667n}$	[Sect. 4.1] SEM+FGDI		
	-	-	$2^{0.643n}$	[Sect. 4.3] SEM+FGDI+IS		
	-	-	$2^{0.625n}$	[Sect. 5.2] SEM+FGMC		
	-	-	$2^{0.612n}$	[Sect. 5.3] SEM+FGMC+IS		
Provable	$2^{0.5n}$	[HS08]	$2^{0.5n}$	[HS08]	$2^{0.5n}$	[HS08]

$\mathcal{H}_1 \parallel \mathcal{H}_2$	Collision		Preimage		2nd Preimage	(challenge length)
Ideal	2^n	Birthday	2^{2n}	Brute-force	2^{2n}	Brute-force
HAIFA	$2^{0.5n}$	[Jou04] MC	2^n	[Jou04] MC		
MD	$2^{0.5n}$	[Jou04] MC	2^n	[Jou04] MC		
	-	-	-	-	$2^{0.75n}$	[Sect. 6.1] SEM+FGDI ($2^{0.75n}$)
	-	-	-	-	$2^{0.735n}$	[Sect. 6.3] SEM+FGDI+IS ($2^{0.735n}$)
Provable	$2^{0.5n}$	[HS08]	$2^{0.5n}$	[HS08]	$2^{0.5n}$	[HS08]

Zipper	Collision		Preimage		2nd Preimage	(challenge length)
Ideal	$2^{0.5n}$	Birthday	2^n	Brute-force	2^n	Brute-force
MD	-	-	-	-	$2^{0.625n}$	[Sect. 7] SEM+FGDI [$L < 2^{n/2}$] ($2^{0.375n}$)
	-	-	-	-	$2^{0.6n}$	[Sect. 7] SEM+FGMC [$L > 2^{n/2}$] ($2^{0.4n}$)
Provable	$2^{0.5r}$	[Lis06]	$2^{0.5r}$	[Lis06]	$2^{0.5r}$	[Lis06]

Hash-Twice	Collision		Preimage		2nd Preimage	(challenge length)
Ideal	$2^{0.5n}$	Birthday	2^n	Brute-force	2^n	Brute-force
MD	-	-	-	-	$2^{0.667n}$	[ABDK09] EM+MC+DS ($2^{0.333n}$)
	-	-	-	-	$2^{0.612n}$	[Sect. 8.3] SEM+IS+DS+FGDI ($2^{0.5n}$)
	-	-	-	-	$2^{0.591n}$	[Sect. 8.2] SEM+IS+DS+FGMC ($2^{0.591n}$)
Provable	$2^{0.5}$	-	2^n	-	$2^{0.5n}$	-

$r : \min(b, n)$ L : 2nd preimage length MC: Joux's Multi-Collisions
EM: Kelsey and Schneier's Expandable Message DS: Diamond Structure
IS: Interchange Structure SEM: Simultaneous Expandable Message
FGDI: Deep-Iterates in Functional Graph FGMC: Multi-Cycles in Functional Graph
All computational complexities of the attacks are presented ignoring the polynomial factors.

Table 1: Security status of various combiners of two narrow-pipe hashes

1.3 Notations and Roadmap of the Rest of Paper

Notations. We summarize below notations shared across various attacks.

$\mathcal{H}_1$, $\mathcal{H}_2$	: Underlying hash functions in a hash combiner
IV_1, IV_2	: Initialization vectors of $\mathcal{H}_1$ and $\mathcal{H}_2$, respectively
h_1, h_2	: Compression functions of $\mathcal{H}_1$ and $\mathcal{H}_2$, respectively
h_1^*, h_2^*	: Compression functions iterated over several blocks (in particular, $\mathcal{H}_i(\boldsymbol{M}) = h_i^*(IV_i, \boldsymbol{M})$ for $i \in \{1, 2\}$)
V	: Targeted image
m	: Message block
M	: Message chunk
$[m]^q$	: Message chunk formed by concatenating q message blocks m, with $[m] = [m]^1$
$M_{\|q}$	: Message chunk with q message blocks
$\boldsymbol{M} = m_1 \| \ldots \| m_L$	: Target message or computed preimage (of L massage blocks)
L	: Length of $\boldsymbol{M}$ (measured in the number of blocks)
ℓ	: The binary logarithm of the length of the message $\boldsymbol{M}$, *i.e.*, denote $L = 2^\ell$
$\boldsymbol{M}'$	: Computed second preimage
L'	: Length of $\boldsymbol{M}'$ (measured in the number of blocks)
$a_0, \ldots, a_L$	: Sequence of internal states computed during the invocation of h_1 on $\boldsymbol{M}$, $a_0 = IV_1$
$b_0, \ldots, b_L$	: Sequence of internal states computed during the invocation of h_2 on $\boldsymbol{M}$, $b_0 = IV_2$
x, y	: Computed internal states
$\vec{a}_j$, $\vec{b}_k$	: Chains of internal states for $\mathcal{H}_1$ and $\mathcal{H}_2$, respectively. $\vec{a}_j$ denotes a generic chain, while $\vec{a}_{j_0}$ denotes a particular chain
A_j, B_k	: End points (final states) of the chains
n	: Bit-size of the output of each underlying hash function ($\mathcal{H}_1$ and $\mathcal{H}_2$); In addition, we suppose their compression functions h_1 and h_2 have n-bit internal states (*i.e.*, suppose the underlying hash functions are narrow-pipe)
b	: Bit-size of a message block
N	: The considered random mappings are from a finite N-set domain to a finite N-set range, and $N = 2^n$
$\mathcal{FG}_{f_1}$, $\mathcal{FG}_{f_2}$	: The functional graphs of random mappings $f_1(\cdot) \triangleq h_1(\cdot, m)$ and $f_2(\cdot) \triangleq h_2(\cdot, m)$ generated by fixing a message block m to the compression functions
$\mathcal{M}$	: A set of messages
$\mathcal{M}_{\texttt{MC/EM/SEM/DS/IS}}$	: The set of messages in a standard Joux's multi-collision or an expandable message or a simultaneous expandable message or a diamond structure or an interchange structure

$x \xrightarrow{m} x'$, $x \xrightarrow{\hat{M}} x'$	:	We say that m (resp. $\hat{M}$) maps state x to state x' if $x' = h(x, m)$ (resp. $x' = h^*(x, \hat{M})$) and denote this by $x \xrightarrow{m} x'$ (resp. $x \xrightarrow{\hat{M}} x'$, the compression function h is clear from the context).
$h_{[m]}$	:	An n-bit random mapping obtained by feeding an arbitrary fixed message block m into a compression function h with n-bit state.
$\tilde{O}$	:	Soft-O, is used as a variant of big O notation that ignores logarithmic factors. Thus, $f(n) \in \tilde{O}(g(n))$ is shorthand for $\exists k : f(n) \in O(g(n) \log^k g(n))$.

Roadmap. Section 2 exhibits preliminaries including generic tools and known attacks on hash constructions. Particularly, sections 2.5, 2.6, 2.7 demonstrate those new tools – the interchange structure, the simultaneous expandable message, deep iterates and multi-cycles in random functional graphs – which make the presented attacks possible. Sections 3, 4, and 5 illustrate preimage attacks on the XOR combiner using interchange structure, deep iterates, and multi-cycles in functional graphs, respectively. Sections 6, 7, and 8 describes the second-preimage attacks on the concatenation combiner, Zipper hash, and Hash-Twice, respectively. Within the description of each attack, we provide an overview followed by detailed steps. In Section 9, we discuss more applications and extensions of the proposed attacks, including applications to combiners of wide-pipe hash functions and extensions to the combination of more than two hash functions. Section 10 summarizes attacks presented in this paper and discusses open problems.

2 Preliminaries

In this section, we introduce the technical tools and general concepts used in our attacks. Those tools briefly introduced in Sect. 2.1 through Sect. 2.4 are all existing and well-known tools. Those tools described in detail in Sect. 2.5 through Sect. 2.7 are our new tools exploited in different attacks.

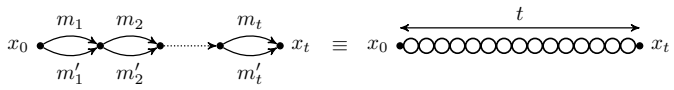
2.1 Joux's Multi-Collision (MC) and Its Applications in Attacks on the Concatenation Combiner [Jou04]

In 2004, Joux introduced multi-collisions on narrow-pipe Merkle-Damgård hash functions. Given a hash function $\mathcal{H}$, a multi-collision refers to a set of messages $\mathcal{M} = \{M_1, M_2, \ldots\}$ whose hash digests are all the same, *i.e.*, $\mathcal{H}(M_i) = \mathcal{H}(M_j)$ for any pair $M_i, M_j \in \mathcal{M}$. The computational complexity of generic brute-force search increases exponentially when the target size $|\mathcal{M}|$ increases; more precisely, it is approximately $2^{(|\mathcal{M}|-1)\cdot n/|\mathcal{M}|}$. Utilizing the iterative nature of Merkle-Damgård structure, Joux's algorithm (see Alg. 1, whose pseudo-code is given in Appendix A) is able to find multi-collision of size 2^t with a complexity of $t \cdot 2^{n/2}$, *i.e.*, a complexity not much greater than that of finding a single collision.

Algorithm 1: Building a 2^t-Joux's Multi-Collision

Require: Given an iterated hash function $\mathcal{H}$ with a compression function h, and an initial value x_0.

1. Initialize $\mathcal{M}_{\texttt{MC}}$ as a data structure of pairs of message blocks
2. For $i = 1, \ldots, t$:
   (a) Find a pair of message blocks (m_i, m_i') such that $h(x_{i-1}, m_i) = h(x_{i-1}, m_i') = x_i$. This can be done with a complexity of $2^{n/2}$ due to birthday paradox.
   (b) Append (m_i, m_i') to $\mathcal{M}_{\texttt{MC}}$
3. Output $(x_t, \mathcal{M}_{\texttt{MC}})$



It is trivial to see the message set $\mathcal{M} = \{\overline{m}_1 \parallel \overline{m}_2 \parallel \cdots \parallel \overline{m}_t \mid \overline{m}_i = m_i \text{ or } m_i' \text{ for } i = 1, 2, \ldots, t\}$ forms a multi-collision of size 2^t, and the overall complexity is $\mathcal{O}(t \cdot 2^{n/2})$. Moreover, a data structure $\mathcal{M}_{\texttt{MC}}$ of t pairs of message blocks can fully define the set $\mathcal{M}$ of 2^t colliding messages.

With Joux's multi-collision at hand, one can immediately deploy a collision attack and a preimage attack on concatenation combiner with complexities $n \cdot 2^{n/2}$ and $n \cdot 2^n$, respectively. The collision attack goes as follows: first, build a $2^{n/2}$-Joux's multi-collision for one of the underlying hash function (iterated), and then exploit the messages in the structure to launch a birthday attack for the other hash function to find a collision among the outputs. The preimage attack follows a similar framework (see [BGW18] for an illustration and Joux's original paper [Jou04] for more details).

Since its invention, Joux's multi-collisions have been employed in numerous cryptanalysis of hash functions, including the following most relevant ones and works such as [HS06,NS07].

2.2 Expandable Message (EM) and the Long Message Second-Preimage Attack [KS05]

In [DA99a], Dean devised a second-preimage attack for long messages on specific Merkle-Damgård hash functions for which it is easy to find fixed points in their compression function. Given a challenge message $\boldsymbol{M} = m_1 \parallel m_2 \parallel \ldots \parallel m_L$, the attacker computes the sequence of internal states $a_0, a_1, \ldots, a_L$ generated during the invocation of the compression function on $\boldsymbol{M}$. A simplified attack would now start from the state $x_0 = IV$ and evaluate the compression function with arbitrary message blocks until a collision $h(x_0, m) = a_p$ is found for some message block m and index p. The attacker can now append the message suffix

$m_{p+1}\|\ldots\|m_L$ to m, hoping to obtain the target hash value $\mathcal{H}(\boldsymbol{M})$. However, this approach does not work due to the final padding of the message length, which will be different if the message prefixes are of different lengths. The solution of Dean was to compute an *expandable message* that consists of the initial state x_0 and another state $\hat{x}$ such that for each length κ (in some range), there is a message $M_{\|\kappa}$ of κ blocks that maps x_0 to $\hat{x}$. Thus, the algorithm first finds a collision $h(\hat{x}, m) = a_p$, and the second preimage is computed as $M_{\|p-1}\|m\|m_{p+1}\|\ldots\|m_L$. The assumption that it is easy to find fixed points in the compression function is used in efficient construction of the expandable message.

In [KS05], Kelsey and Schneier described a more generic attack that uses multi-collisions of a special form to construct an expandable message, removing the restriction of Dean regarding fixed points. As in Joux's original algorithm, the multi-collisions are constructed iteratively in t steps. In the i-th step, we find a collision between some m_i and m'_i such that $|m_i| = 1$ (it is a single block) and $|m'_i| = 2^{i-1}+1$, namely, $h(x_{i-1}, m_i) = h(x_{i-1}, m'_i)$. This is done by firstly picking an arbitrary prefix of size 2^{i-1} of m'_i denoted by $\hat{m}_i$, say $[0]^{2^{i-1}}$, computing $h(x_{i-1}, \hat{m}_i) = x'_i$ and then looking for a collision $h(x_{i-1}, m_i) = h(x'_i, \check{m}_i)$ using a final block $\check{m}_i$ (namely, $m'_i = \hat{m}_i \,\|\, \check{m}_i$) (see Fig. 6).

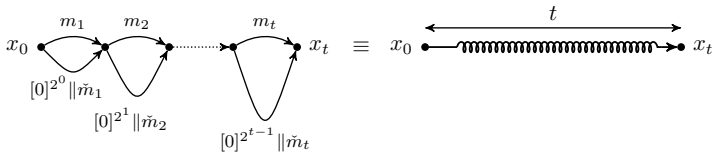


Fig. 6: The expandable message and its condensed representation [JN15]

The construction of Kelsey and Schneier gives an expandable message that can be used to generate messages starting from x_0 and reaching $\hat{x} = x_t$ whose (integral) sizes are in the interval $[t, 2^t+t-1]$ (such a message structure is denoted as a $(t, 2^t + t - 1)$-expandable message). A message of length $t \leq \kappa \leq 2^t + t - 1$ is generated by looking at the t-bit binary representation of $\kappa - t$. In iteration $i \in \{1, 2, \ldots, t\}$, we select the long message fragment m'_i if the i-th LSB of $\kappa - t$ is set to 1 (otherwise, we select the single block m_i). In the sequel, we denote this type of expandable message by $\mathcal{M}_{\texttt{EM}}$. Given that the challenge message $\boldsymbol{M}$ is of $L \leq 2^{n/2}$ blocks, the construction of the expandable message in the first phase of the attack requires less than $n \cdot 2^{n/2}$ computations, while obtaining the collision with one of the states computed during the computation of $\boldsymbol{M}$ requires approximately $1/L \cdot 2^n$ computations according to the birthday paradox.

2.3 Diamond Structure (DS) [KK06]

Like Joux's multi-collisions and expandable message, the diamond structure is also a type of multi-collision. The difference is that instead of mapping a common starting state to a final state, each message in a diamond maps a different state

to a final state. A 2^t-diamond contains 2^t specially structured messages mapping 2^t starting states to a final state, and it forms a complete binary tree of depth t. The 2^t starting states are leaves, and the final state is the root. A 2^t-diamond can be built by launching several collision attacks requiring about $\sqrt{t} \cdot 2^{\frac{(n+t)}{2}}$ messages and $n \cdot \sqrt{t} \cdot 2^{\frac{(n+t)}{2}}$ computations in total [BSU12]. In the sequel, we denote the set of messages in a diamond by $\mathcal{M}_{\mathtt{DS}}$. The diamond was primarily invented by Kelsey and Kohno to devise herding attacks against MD hash functions [KK06], in which the attacker first commits to the digest value of a message using the root of his diamond and later "herds" any given prefix of a message to his commitment by choosing an appropriate message from his diamond as the suffix. Later, Andreeva *et al.* successfully exploited it to launch herding and/or second-preimage attack beyond MD hash constructions, such as the dithered hash, Hash-Twice, the Zipper hash, and hash trees [ABF+08,ABDK09,ABD+16]. Concretely, the second-preimage attack on Hash-Twice in [ABDK09] leverages techniques in herding attack and techniques in the above-mentioned second-preimage attack. One key point of this attack is that it builds a long Joux's multi-collision in the first pass, exploits messages in this multi-collision to build a diamond structure in the second pass, and finally uses the diamond as a connector to connect one crafted message to the challenge message on some states. Let 2^t be the width of the diamond and 2^ℓ be the length of the message; the complexity of this attack is approximately $2^{(n+t)/2} + 2^{n-\ell} + 2^{n-t}$.

2.4 Distinguished Points (DP)

The memory complexity of many algorithms that are based on functional graphs (*e.g.*, parallel collision search [vOW99]) can be reduced by utilizing the *distinguished points* method (which is attributed to Ron Rivest). Assume that our goal is to detect a collision of a chain (starting from an arbitrary node) with the nodes of $\mathcal{G}$ computed in Alg. 5, but without storing all the 2^t nodes in memory. The idea is to define a set of 2^t distinguished points (nodes) using a simple predicate (*e.g.*, the $n-t$ LSBs of a node are zero). The nodes of $\mathcal{G}$ contain approximately $2^t \cdot 2^{t-n} = 2^{2t-n}$ distinguished points, and only they are stored in memory. A collision of an arbitrary chain with $\mathcal{G}$ is expected to occur at depth of about 2^{n-t} and will be detected at the next distinguished point which is located (approximately) after traversing additional 2^{n-t} nodes. Consequently, we can detect the collision with a small overhead in time complexity, but a significant saving factor of 2^{n-t} in memory.

Interestingly, in the specific attack of Sect. 4, the distinguished points method is essential for reducing the time complexity of the algorithm.

2.5 Interchange Structure (IS)

In this subsection, we present how to build a structure that enables us to simultaneously control two (or more) hash computation lanes sharing the same input message and succeed in further relaxing the pairwise relation between the

internal states of computation lanes. We name the structure the *interchange structure*.

The main idea is to consider several chains of internal states reached by processing a common message $\boldsymbol{M}$ from different starting points (note that the message $\boldsymbol{M}$ is not fixed in advance, but will be determined when building the structure). More precisely, the message $\boldsymbol{M}$ is denoted as the *primary* message and divided into several chunks: $\boldsymbol{M} = M_0 \,\|\, M_1 \,\|\, \ldots$ (as discussed later, a chunk consists of approximately $n/2$ message blocks). We denote chains of internal states for $\mathcal{H}_1$ as $\vec{a}_j$ and the individual states of the chain as $\vec{a}_j^i$, with $h_1^*(\vec{a}_j^i, M_i) = \vec{a}_j^{i+1}$. Similarly, we denote chains for $\mathcal{H}_2$ as $\vec{b}_k$, with $h_2^*(\vec{b}_k^i, M_i) = \vec{b}_k^{i+1}$. When considering both hash functions, message block M_i leads from the pair of states $(\vec{a}_j^i, \vec{b}_k^i)$ to $(\vec{a}_j^{i+1}, \vec{b}_k^{i+1})$, which is denoted as

$$(\vec{a}_j^i, \vec{b}_k^i) \overset{M_i}{\rightsquigarrow} (\vec{a}_j^{i+1}, \vec{b}_k^{i+1}).$$

Switch Structure. To construct a desired interchange structure, we first create the basic building blocks to jump between chains in a controlled way; we named them *switches*. A switch allows to jump from a specific pair of chains $(\vec{a}_{j_0}, \vec{b}_{k_0})$ to a different pair of chains $(\vec{a}_{j_0}, \vec{b}_{k_1})$ using a secondary message chunk M_i', in addition to the normal transitions using chunk M_i of the primary message $\boldsymbol{M}$:

$$(\vec{a}_j^i, \vec{b}_k^i) \overset{M_i}{\rightsquigarrow} (\vec{a}_j^{i+1}, \vec{b}_k^{i+1}) : \quad \text{normal transition for each chain}$$

$$(\vec{a}_{j_0}^i, \vec{b}_{k_0}^i) \overset{M_i'}{\rightsquigarrow} (\vec{a}_{j_0}^{i+1}, \vec{b}_{k_1}^{i+1}) : \quad \text{jump from chains } (\vec{a}_{j_0}, \vec{b}_{k_0}) \text{ to } (\vec{a}_{j_0}, \vec{b}_{k_1})$$

To simplify the notation, we often omit the chunk index to show only the chains that are affected by the switch.

The main message chunk M_i and the secondary message chunk M_i' are determined when building the switch, and the main message defines the next state of all the chains. We note that the secondary message chunk M_i' should only be used when the state is $(\vec{a}_{j_0}^i, \vec{b}_{k_0}^i)$. A simple example is depicted in Fig. 7.

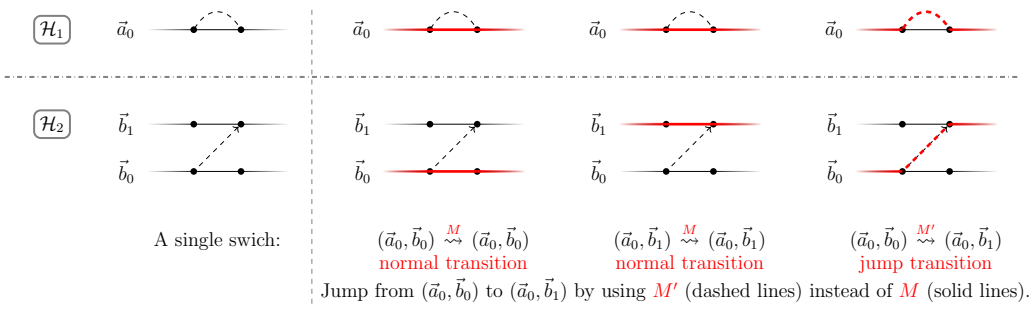


Fig. 7: A single switch

Alternatively, a switch can be designed to jump from $(\vec{a}_{j_0}, \vec{b}_{k_0})$ to $(\vec{a}_{j_1}, \vec{b}_{k_0})$. It can be built with a complexity of $\tilde{O}(2^{n/2})$.

We now explain how to build the switch structure at the core of some of our attacks. This construction is strongly based on the multi-collision technique of Joux presented in Sect.2.1.

Given states $\vec{a}^i_{j_0}$, $\vec{b}^i_{k_0}$ and $\vec{b}^i_{k_1}$, we want to build message chunks M_i and M'_i in order to have the following transitions:

$$\begin{aligned}
(\vec{a}^i_{j_0}, \vec{b}^i_{k_0}) &\overset{M_i}{\rightsquigarrow} (\vec{a}^{i+1}_{j_0}, \vec{b}^{i+1}_{k_0}): && \text{normal transition}\\
(\vec{a}^i_{j_0}, \vec{b}^i_{k_1}) &\overset{M_i}{\rightsquigarrow} (\vec{a}^{i+1}_{j_0}, \vec{b}^{i+1}_{k_1}): && \text{normal transition}\\
(\vec{a}^i_{j_0}, \vec{b}^i_{k_0}) &\overset{M'_i}{\rightsquigarrow} (\vec{a}^{i+1}_{j_0}, \vec{b}^{i+1}_{k_1}): && \text{jump transition}
\end{aligned}$$

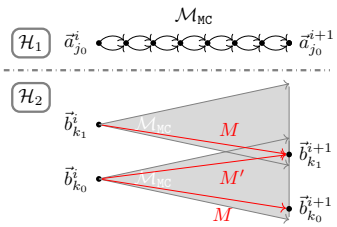
The main message chunk M_i is used to define the next state of all the remaining chains, while the secondary message chunk M'_i will be used to jump from chains $(\vec{a}_{j_0}, \vec{b}_{k_0})$ to $(\vec{a}_{j_0}, \vec{b}_{k_1})$. We note that M'_i will only be used when the state is $(\vec{a}^i_{j_0}, \vec{b}^i_{k_0})$. In particular, M_i and M'_i must satisfy the following:

$$\begin{aligned}
\vec{a}^{i+1}_{j_0} &= h^*_1(\vec{a}^i_{j_0}, M_i) = h^*_1(\vec{a}^i_{j_0}, M'_i)\\
\vec{b}^{i+1}_{k_1} &= h^*_2(\vec{b}^i_{k_1}, M_i) = h^*_2(\vec{b}^i_{k_0}, M'_i)\\
\vec{b}^{i+1}_{k_0} &= h^*_2(\vec{b}^i_{k_0}, M_i) \neq \vec{b}^{i+1}_{k_1}
\end{aligned}$$

The full building procedure is shown in Alg. 2 whose pseudo-code is given in Appendix A; it requires approximately $n/2 \cdot 2^{n/2}$ evaluations of the compression functions.

Algorithm 2: Building a single switch

1. Build a multi-collision for h^*_1, starting from state $\vec{a}^i_{j_0}$, *i.e.*, a set $\mathcal{M}_{\texttt{MC}}$ of $2^{n/2}$ messages that all reach the same state $\vec{a}^{i+1}_{j_0}$ ($\forall M \in \mathcal{M}_{\texttt{MC}}$, $h^*_1(\vec{a}^i_{j_0}, M) = \vec{a}^{i+1}_{j_0}$). As shown in Sect. 2.1, this can be done efficiently by sequentially building $n/2$ collisions. Thus, each M is comprised of $n/2$ message blocks.
2. Evaluate $h^*_2(\vec{b}^i_{k_0}, M)$ and $h^*_2(\vec{b}^i_{k_1}, M)$ for all the messages M in the set $\mathcal{M}_{\texttt{MC}}$. With high probability there is match between the sets of values[a]. Denote the colliding messages as M_i and M'_i, so that we have $h^*_2(\vec{b}^i_{k_0}, M'_i) = h^*_2(\vec{b}^i_{k_1}, M_i)$.
3. Compute the missing chains using the message M_i: $\vec{a}^{i+1}_j = h^*_1(\vec{a}^i_j, M_i)$, $\vec{b}^{i+1}_k = h^*_2(\vec{b}^i_k, M_i)$. With high probability all the chains reach distinct values; if this is not the case, restart the construction with a new multi-collision.[b]



First, M and M' are selected from $\mathcal{M}_{\text{MC}}$ to generate a collision (defining the new $\vec{b}_{k_1}$), then $\vec{b}_{k_0}$ is evaluated using M.

[a] If this is not the case, we build a new multi-collision.
[b] We omit the illustration of step 3 for simplicity in the figure.

Interchange Structure. By combining several simple switches, we can build an interchange structure with starting points IV_1 and IV_2 and ending points $\{A_j \mid j = 0 \dots 2^t - 1\}$ and $\{B_k \mid k = 0 \dots 2^t - 1\}$, so that we can select a message ending in any state (A_j, B_k). An interchange structure with 2^t chains for each function requires about 2^{2t} switches. Since we can build a switch for a cost of $\tilde{O}(2^{n/2})$, the total structure is built with $\tilde{O}(2^{2t+n/2})$ operations.

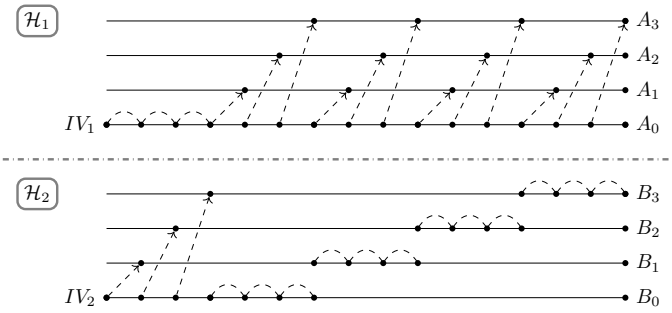
Let us now describe the combination of switch structures into an interchange structure. The goal of this structure is to select the final value of the $\mathcal{H}_1$ computation and the $\mathcal{H}_2$ computation independently. More precisely, the structure defines two sets of final values A_j and B_k, and a set of messages $\boldsymbol{M}_{jk}$ such that

$$(IV_1, IV_2) \overset{\boldsymbol{M}_{jk}}{\rightsquigarrow} (A_j, B_k).$$

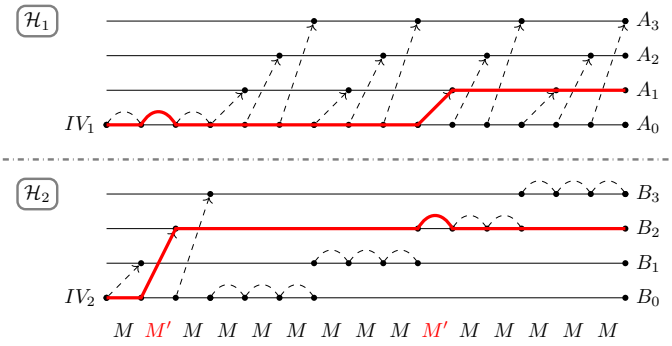
Algorithm 3: Building and using a 2^t-interchange structure

1. Initialize the first chains with $\vec{a}^0_0 = IV_1$, $\vec{b}^0_0 = IV_2$, and set the other starting points randomly.
2. Build switches to jump for an already reachable pair $(\vec{a}_{j_0}, \vec{b}_{k_0})$ to a different pair $(\vec{a}_{j_0}, \vec{b}_{k_1})$ (or to $(\vec{a}_{j_1}, \vec{b}_{k_0})$, respectively). By using $2^{2t} - 1$ switches, we can make all pairs reachable. There are many ways to combine the switches; a simple one can be described as follows:
   (a) first, build switches from $(\vec{a}_0, \vec{b}_0)$ to each of the $(\vec{a}_0, \vec{b}_k)$'s;
   (b) then, for each k, build a series of switches from $(\vec{a}_0, \vec{b}_k)$ to all the $(\vec{a}_j, \vec{b}_k)$'s.

   To reach the chains $(\vec{a}_j, \vec{b}_k)$, one would activate the k-th switch in the first part to jump from $(\vec{a}_0, \vec{b}_0)$ to $(\vec{a}_0, \vec{b}_k)$, and then the j-th switch in the k-th series of the second part to jump from $(\vec{a}_0, \vec{b}_k)$ to $(\vec{a}_j, \vec{b}_k)$.



One possible way to build an interchange structure



Select a given message in the interchange structure

Algorithm 3 describes the combination of switches to build an interchange structure. Its pseudo-code is given in Appendix A, where the INTERCHANGE functions builds the structure, and the SELECTMESSAGE function extracts the message reaching $(\vec{a}_j, \vec{b}_k)$.

The structure can be somewhat optimized using the fact that the extra chains have no prespecified initial values. We show how to take advantage of this in Appendix B, using multi-collision structures in addition to the switch structures. However, this does not significantly change the complexity: we need $(2^t-1)(2^t-1)$ switches instead of $2^{2t}-1$. In total, we need approximately $n/2 \cdot 2^{2t+n/2}$ evaluations of the compression functions to build a 2^t-interchange structure.

We believe that a 2^t-interchange structure based on switches will need at least $\Theta(2^{2t})$ switches, because every switch can only increase the number of reachable pairs $(\vec{a}_j, \vec{b}_k)$ by one. As shown in Appendix B some switches can be saved in the beginning, but it seems that new ideas are needed to reduce the total complexity below $\Theta(2^{2t+n/2})$.

2.6 Simultaneous Expandable Messages (SEM)

In this subsection, we build a *simultaneous expandable message* for two MD hash functions basing on the multi-collision described in Sect. 2.1 and the expandable message for a single MD hash function described in Sect. 2.2. This expandable message consists of the initial states (IV_1, IV_2) and final states $(\hat{x}, \hat{y})$ such that

for each length κ in some appropriate range (determined below), there is a message $M_{\|\kappa}$ of κ blocks that maps (IV_1, IV_2) to $(\hat{x}, \hat{y})$. A similar construction of an expandable message over two hash functions was proposed in the independent paper [JN15] by Jha and Nandi, which analyses the Zipper hash assuming weak compression functions. We describe our construction approach of this simultaneous expandable message in detail next.

We set $C \approx n/2 + \log(n)$ as a parameter that depends on the state size n. Our basic building block consists of two pairs of states (x_0, y_0) and (x_1, y_1) and two message fragments ms and ml that map the state pair (x_0, y_0) to (x_1, y_1). The message ms is the (shorter) message fragment of fixed size C, while ml is of size $i > C$. We will show how to construct this building block for any state pair (x_0, y_0) and length $i > C$ in Alg.4.

Given this building block and a positive parameter t, we build an expandable message in the range of $[C(C-1)+tC, C^2-1+C(2^t+t-1)]$. This is done by utilizing a sequence of $C-1+t$ basic building blocks. The first $C-1$ building blocks are built with parameters $i \in \{C+1, C+2, \ldots, 2C-1\}$. It is easy to see that these structures give a $(C(C-1), C^2-1)$–expandable message by selecting at most one longer message fragment from the sequence, where the remaining $C-2$ (or $C-1$) fragments are of length C. The final t building blocks give a standard expandable message, but it is built in intervals of C. These t building blocks are constructed with parameters $i = C(2^{j-1}+1)$ for $j \in \{1, \ldots, t\}$. See Fig.8 for a visual illustration.

Given a length κ in the range of $[C(C-1)+tC, C^2-1+C(2^t+t-1)]$, we can construct a corresponding message by first computing κ (modulo C). We then select the length $\kappa' \in [C(C-1), C^2-1]$ such that $\kappa' \equiv \kappa$ (modulo C), defining the first $C-1$ message fragment choices. Finally, we compute $(\kappa-\kappa')/C$, which is an integer in the range of $[t, 2^t+t-1]$, and select the final t message fragment choices as in a standard expandable message using the binary representation of $(\kappa-\kappa')/C$.

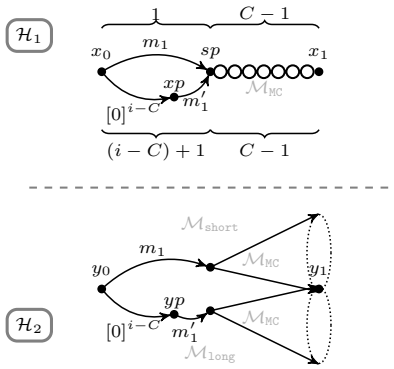
Construction of the Building Block. Given state pair (x_0, y_0) and length $i > C$, the algorithm for constructing the building block for the expandable message is based on multi-collisions, as described below; its pseudo-code is given in Appendix A.

Algorithm 4: Constructing a building block for an SEM

1. Pick an arbitrary prefix of size $i-C$ blocks, say $[0]^{i-C}$, and compute $xp = h_1^*(x_0, [0]^{i-C})$.
2. Find collision (m_1, m_1', sp) s.t. $h_1(x_0, m_1) = h_2(xp, m_1') = sp$, where m_1 and m_1' are single message blocks.
3. Build a 2^{C-1} standard Joux's multi-collision in h_1 starting from sp, and denote its endpoint by x_1. Altogether, we have a multi-collision in h_1 with 2^C messages that map x_0 to x_1. Out of these 2^C messages,

- 2^{C-1} are of length C (obtained by first selecting m_1, *i.e.*, $m_1 \times \mathcal{M}_{\texttt{MC}}$), and we denote this set of short messages by $\mathcal{M}_{\texttt{short}}$;
- 2^{C-1} are of length i (obtained by first selecting $[0]^{i-C} \parallel m'_1$, *i.e.*, $([0]^{i-C} \parallel m'_1) \times \mathcal{M}_{\texttt{MC}}$), and we denote this set of long messages by $\mathcal{M}_{\texttt{long}}$.

4. Evaluate $yp = h_2^*(y_0, [0]^{i-C})$ and store the result. Next, evaluate h_2 from y_0 on the two sets $\mathcal{M}_{\texttt{short}}$ and $\mathcal{M}_{\texttt{long}}$ (using the stored yp to avoid recomputing $h_2^*(y_0, [0]^{i-C})$) and find a collision between them (such a collision is very likely to occur since $C - 1 > n/2$). The collision gives the required $ms \in \mathcal{M}_{\texttt{short}}$ and $ml \in \mathcal{M}_{\texttt{long}}$ of appropriate sizes such that $y_1 \triangleq h_2^*(y_0, ms) = h_2^*(y_0, ml)$ and $x_1 \triangleq h_1^*(x_0, ms) = h_1^*(x_0, ml)$.



The complexity of Step 1 is less than i compression function evaluations. The complexity of Step 2 is approximately $2^{n/2}$, while the complexity of Step 3 is approximately $C \cdot 2^{n/2} \approx n \cdot 2^{n/2}$. The complexity of Step 4 is approximately $i + n \cdot 2^{n/2}$. In total, the complexity of constructing the basic building block is approximately $i + n \cdot 2^{n/2}$ (ignoring small factors).

Complexity Analysis of the Full Building Procedure. The full expandable message requires computing $C - 1 + t$ building blocks whose sum of length parameters (dominated by the final building block) is approximately $C \cdot 2^t \approx n \cdot 2^t$. Assuming that $t < n$, we construct $C - 1 + t \approx n$ building blocks, and the total time complexity of constructing the expandable message is approximately $n \cdot 2^t + n^2 \cdot 2^{n/2}$. Our attacks require the $(C(C-1)+tC, C^2-1+C(2^t+t-1))$–expandable message to extend up to length L, implying that $L \approx n \cdot 2^t$ and giving a time complexity of approximately

$$L + n^2 \cdot 2^{n/2}.$$

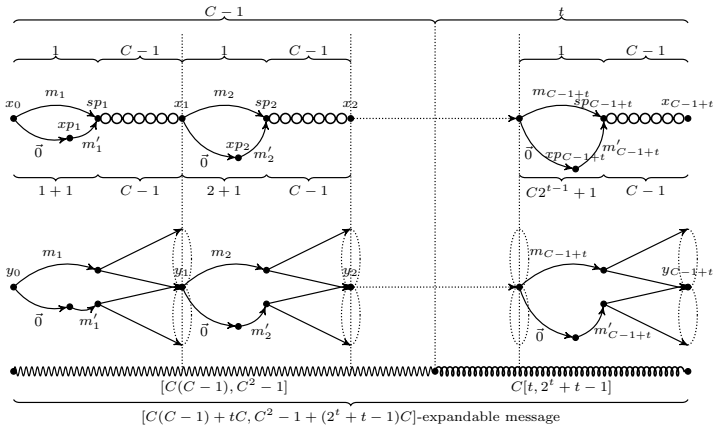


Fig. 8: Simultaneous expandable message

2.7 Functional Graph (FG) of Random Mappings

In many of our attacks, we evaluate a compression function h with a fixed message input block m (*e.g.*, the zero block) and simplify our notation by defining $f(x) = h_{[m]}(x) = h(x, m)$. The mapping f yields a directed functional graph.

The functional graph of a random mapping f is defined via successive iteration on this mapping.

> Let f be an element in $\mathcal{F}_N$ that is the set of all mappings with the set N as both the domain and range. The functional graph of f is a directed graph whose nodes are the elements $0, \ldots, N-1$ and whose edges are the ordered pairs $\langle x, f(x)\rangle$, for all $x \in \{0, \ldots, N-1\}$. If starting from any x_0 and iterating f, that is $x_1 = f(x_0), x_2 = f(x_1), \ldots$, we will find that before N iterations, a value x_j equal to one of $x_0, x_1, \ldots, x_{j-1}$; suppose that the collided one is x_i. In this case, we say the path $x_0 \to x_1 \to \cdots \to x_{i-1} \to x_i$ connects to a *cycle* $x_i \to x_{i+1} \to \cdots \to x_{j-1} \to x_i$. If we consider all possible starting points x_0, paths exhibit confluence and form trees; trees grafted on cycles form components; a collection of components forms a functional graph. That is, a functional graph can be viewed as a set of connected components; a component is a cycle of trees; a tree is recursively defined by appending a node to a set of trees; a node is a basic atomic object and is labelled by an integer [FO89].

Structures of functional graph of random mappings have been studied for a long time, and some parameters have accurate asymptotic evaluations [FO89]. Below, we list some of the most relevant ones. These properties have been extensively studied and exploited in cryptography, *e.g.*, in the classical works of Hellman [Hel80] and van Oorschot and Wiener [vOW99], and much more recently in generic attacks on hash-based MACs [DL14,GPSW14,LPW13,PK14,PW14] (refer to [BGW18] for a systematization of knowledge regarding the applications of random functional graphs in generic attacks).

Theorem 1 ([FO89]). *The expected number of components, number of cyclic nodes (nodes belong to a cycle), number of terminal nodes (nodes without preim-*

age: $f^{-1}(x) = \emptyset$), number of image nodes (nodes with preimage), and number of k-th iterate image nodes (image nodes of the k-th iterate f^k of f) in a random mapping of size N have the following asymptotic forms as $N \to \infty$:

1. *# Components* $\frac{1}{2}\log N = 0.5 \cdot n$
2. *# Cyclic nodes* $\sqrt{\pi N/2} \approx 1.2 \cdot 2^{n/2}$
3. *# Terminal nodes* $e^{-1}N \approx 0.37 \cdot 2^n$
4. *# Image nodes* $(1-e^{-1})N \approx 0.62 \cdot 2^n$
5. *# k-th iterate image nodes* $(1-\tau_k)N$, *where* τ_k *satisfies the recurrence relation* $\tau_0 = 0$, $\tau_{k+1} = e^{-1+\tau_k}$

Seen from an arbitrary node x_0, we call the length (measured by the number of edges) of the path starting from x_0 and before entering a cycle the *tail length* of x_0 and denote it by $\lambda(x_0)$; term the length of the cycle connected with x_0 the *cycle length* of x_0 and denote it by $\mu(x_0)$; name the length of the non-repeating trajectory of the node x_0 the *rho-length* of x_0 and denote it by $\rho(x_0) = \lambda(x_0) + \mu(x_0)$.

Theorem 2 ([FO89]). *Seen from a random point (any of the N nodes in the associated functional graph is taken equally likely) in a random mapping of $\mathcal{F}_N$, the expected tail length, cycle length, rho-length have the following asymptotic forms:*

1. *Tail length* (λ) $\sqrt{\pi N/8} \approx 0.62 \cdot 2^{n/2}$
2. *Cycle length* (μ) $\sqrt{\pi N/8} \approx 0.62 \cdot 2^{n/2}$
3. *Rho-length* (ρ) $\sqrt{\pi N/2} \approx 1.2 \cdot 2^{n/2}$

Theorem 3 ([FO89]). *The expected maximum cycle length (μ^{max}), maximum tail length (λ^{max}) and maximum rho length (ρ^{max}) in the functional graph of a random mapping of $\mathcal{F}_N$ satisfy the following:*

1. $\mathbf{E}\{\mu^{max} \mid \mathcal{F}_N\} = 0.78248 \cdot 2^{n/2}$
2. $\mathbf{E}\{\lambda^{max} \mid \mathcal{F}_N\} = 1.73746 \cdot 2^{n/2}$
3. $\mathbf{E}\{\rho^{max} \mid \mathcal{F}_N\} = 2.41490 \cdot 2^{n/2}$

Theorem 4 ([FO89]). *Assuming the smoothness condition, the expected value of the size of the largest tree and the size of the largest connected component in a random mapping of $\mathcal{F}_N$ are asymptotically*

1. *Largest tree:* $0.48 \cdot 2^n$
2. *Largest component:* $0.75782 \cdot 2^n$

The results from these theorems indicate that in a random mapping, most of the points tend to be grouped together in a single giant component. This component is therefore expected to have very tall trees and a large cycle [FO89].

A useful algorithm for expanding the functional graph of f is given below (see Alg. 5 whose pseudo-code is given in Appendix A). This algorithm is not new and has been previously used (for example, in [GPSW14,PW14]). It takes an input parameter $t \geq n/2$ that determines the number of expanded nodes (and the running time of the algorithm).

Algorithm 5: Expanding the functional graph of f

1. Initialize $\mathcal{G} = \emptyset$ as a data structure of evaluated nodes.
2. Until $\mathcal{G}$ contains 2^t nodes:

(a) Pick an arbitrary starting point x_0 and evaluate the chain $x_{i+1} = f(x_i)$ until it cycles (there exists $x_i = x_j$ for $i \neq j$) or hits a point in $\mathcal{G}$. Add the points of the chain to $\mathcal{G}$.

Deep Iterates in the Functional Graphs (FGDI). Next, we describe our observations on functional graph of random mappings. The efficiencies of our following attacks are mostly based on these observations on special nodes in functional graphs.

In our attacks, we are particularly interested in nodes of f that are located deep in the functional graph. More specifically, x' is an iterate of depth i if there exists some ancestor node x such that $x' = f^i(x)$, *i.e.*, x' is an i-th iterate image node (or say *i-th iterate* for short). If i is relatively large, we say that x' is a deep iterate. Deep iterates are usually obtained using *chains* evaluated from an arbitrary starting point x_0 by computing a sequence of nodes using the relation $x_{i+1} = f(x_i)$. We denote this sequence by $\vec{x}$. The following two observations regarding deep iterates make them helpful in the proposed attacks:

Observation 1. It is easy to obtain a large set of deep iterates. Specifically, by running Alg. 5 with input parameter t ($t \geq n/2$), one can obtain a set of 2^t nodes, among which a constant fraction ($\Theta(2^t)$) are 2^{n-t}-th iterates. The theoretical reasoning is as follows. After we have executed the algorithm and developed 2^t nodes, then another chain from an arbitrary starting point is expected to collide with the evaluated graph at depth of roughly 2^{n-t}. This is a direct consequence of the birthday paradox. Moreover, for two chains from two different starting points x and y, the probability that $\Pr[f^{2^{n-t}}(x) = f^{2^{n-t}}(y)] = \Theta(2^{-t})$ [DL14, Lemma 1] (note that $n-t < n/2$). That is, for $t \geq n/2$, when the number of new chains (of length 2^{n-t} and from arbitrary starting points) is less than 2^t, they are expected to collide with the evaluated graph at distinct points. In particular, this observation implies that most chains developed by the algorithm will be extended to depth $\Omega(2^{n-t})$ (without colliding with $\mathcal{G}$ of cycling); therefore, a constant fraction of the developed nodes are iterates of depth 2^{n-t}. In total, the algorithm develops $\Theta(2^t)$ iterates of f of depth 2^{n-t} in 2^t time. This conclusion was also verified experimentally.

Observation 2. A deep iterate has a relatively high probability to be encountered during the evaluation of a chain from an arbitrary starting node. Let f_1 and f_2 be two independent n to n-bit mappings. Suppose $\bar{x}$ (resp. $\bar{y}$) is an iterate of depth 2^g in $\mathcal{FG}_{f_1}$ (resp. $\mathcal{FG}_{f_2}$); then, it is an endpoint of a chain of states of length 2^g. Let d be in the interval $[1, 2^g]$ and x_0 (resp. y_0) be a random point. Then, according to Lemma 1, $\Pr[x_d = \bar{x} \approx d \cdot 2^{-n}]$ (resp. $\Pr[y_d = \bar{y} \approx d \cdot 2^{-n}]$), which is the probability that $\bar{x}$ (resp. $\bar{y}$) will be encountered at distance d from x_0 (resp. y_0). Due to the independence of f_1 and f_2, $\Pr[x_d = \bar{x} \bigwedge y_d = \bar{y}] \approx (d \cdot 2^{-n})^2$. Summing the probabilities of the (disjoint) events over all distances d in the interval $[1, 2^g]$, we conclude that the probability that $\bar{x}$ and $\bar{y}$ will be encountered at the same distance is approximately $(2^g)^3 \cdot 2^{-2n} = 2^{3g-2n}$.

The probability calculation in Observation 2 yields the conclusion that we need to compute approximately 2^{2n-3g} chains from different starting points to find a pair of starting points (x_0, y_0) reaching a pair of 2^g-th iterates $(\bar{x}, \bar{y})$ at the same distance. This conclusion was verified experimentally. Note that since various trials performed by selecting different starting points for the chains are dependent, the proof of this conclusion is incomplete. However, this dependency is negligible in our attacks, and thus we can ignore it. More details can be found in Appendix C.

Lemma 1. *Let f be an n-bit random mapping and x'_0 an arbitrary point. Let $D \leq 2^{n/2}$ and define the chain $x'_i = f(x'_{i-1})$ for $i \in \{1, \ldots, D\}$ (namely, x'_D is an iterate of depth D). Let x_0 be a randomly chosen point, and define $x_d = f(x_{d-1})$ for integer $d \geq 1$. Then, for any $d \in \{1, \ldots, D\}$, $\Pr[x_d = x'_D] = \Theta(d \cdot 2^{-n})$.*

Proof. (Sketch.) We can assume that the chains do not cycle (*i.e.*, each chain contains distinct nodes), as $D \leq 2^{n/2}$. For $x_d = x'_D$ to occur, x_{d-i} should collide with x'_{D-i} for[14] some $0 \leq i \leq d$. For a fixed i, the probability for this collision is roughly[15] 2^{-n}, and summing over all $0 \leq i \leq d$ (all events are disjointed), we get that the probability is approximately $d \cdot 2^{-n}$. □

Multi-Cycles in Functional Graphs (FGMC) Next, we study a property of some more special nodes — *cyclic nodes* in random functional graphs. There are efficient cycle search algorithms (with $O(2^{n/2})$ time complexity) to detect the cycle length and collect cyclic nodes in the largest component of a random functional graph [Jou09, Chapter 7], and cycles has been exploited in generic attacks on hash-based MACs [GPSW14,LPW13]. Here, we exploit them in a new way. Each cyclic node in a functional graph defined by f loops along the cycle when computed by f iteratively and goes back to itself after a (multi-) cycle-length number of function calls. This property can be utilized to provide extra degrees of freedom when estimating the distance of other nodes to a cyclic node in the functional graph, *i.e.*, it can be expanded to a set of discrete values by using multi-cycles. For example, let x and x' be two nodes in a component of the functional graph defined by f, x be a cyclic node, and the cycle length of the component be denoted as L. Clearly, there exists a path from x' to x as they are in the same component, and the path length is denoted by d. Then, we have the following:

$$f^d(x') = x; \quad f^L(x) = x \quad \Longrightarrow \quad f^{(d+i\cdot L)}(x') = x \quad \text{for any positive integer } i.$$

Suppose it is limited to use at most t cycles (limitation imposed by the length of the message). Then, the distance from x' to x is expanded to a set of $t+1$ values $\{d + i \cdot L \mid i = 0, 1, 2, ..., t\}$.

[14] A collision between x_{d-i} and x'_{D-i} occurs if $x_{d-i} = x'_{D-i}$ but $x_{d-i-1} \neq x'_{D-i-1}$.

[15] A more accurate analysis would take into account the event that the chains collide before x_{d-i}, but the probability of this is negligible.

Now, let us consider a special case of reaching two deep iterates from two random starting nodes: *select two cyclic nodes within the largest components in the functional graphs as the deep iterates*. More specifically, let $\mathcal{FG}_{f_1}$ and $\mathcal{FG}_{f_2}$ be two functional graphs defined by f_1 and f_2. Let $\bar{x}$ and x_0 be two nodes in a common largest component of $\mathcal{FG}_{f_1}$, where $\bar{x}$ is a cyclic node. Let L_1 denote the cycle length of the component and d_1 denote the path length from x_0 to $\bar{x}$. Similarly, we define notations $\bar{y}$, y_0, L_2 and d_2 in $\mathcal{FG}_{f_2}$. We are interested in the probability of linking x_0 to $\bar{x}$ and y_0 to $\bar{y}$ at a common distance. Thanks to the usage of multiple cycles, the distance values from x_0 to $\bar{x}$ and from y_0 to $\bar{y}$ can be selected from two sets $\{d_1 + i \cdot L_1 \mid i = 0, 1, 2, \ldots, t\}$ and $\{d_2 + j \cdot L_2 \mid j = 0, 1, 2, \ldots t\}$, respectively. Hence, as long as there exists a pair of integers (i, j) such that $0 \leq i, j \leq t$ and $d_1 + i \cdot L_1 = d_2 + j \cdot L_2$, we obtain a common distance $d = d_1 + i \cdot L_1 = d_2 + j \cdot L_2$ such that

$$f_1^d(x_0) = \bar{x}, \qquad f_2^d(y_0) = \bar{y}.$$

Next, we evaluate the probability amplification of reaching $(\bar{x}, \bar{y})$ from a random pair (x_0, y_0) at the same distance. Without loss of generality, we assume $L_1 \leq L_2$. Let $\Delta L \triangleq L_2 \mod L_1$. Then, it follows that

$$\begin{aligned} d_1 + i \cdot L_1 &= d_2 + j \cdot L_2 && \Longrightarrow \\ d_1 - d_2 &= j \cdot L_2 - i \cdot L_1 && \Longrightarrow \\ (d_1 - d_2) \mod L_1 &= j \cdot \Delta L \mod L_1 \end{aligned}$$

Letting j range over all integer values in internal $[0, t]$, we will collect a set of $t + 1$ values $\mathcal{D} = \{j \cdot \Delta L \mod L_1 \mid j = 0, 1, \ldots, t\}$.[16] Since $d_1 = \mathcal{O}(2^{n/2})$, $d_2 = \mathcal{O}(2^{n/2})$ and $L_1 = \Theta(2^{n/2})$, it follows that $|d_1 - d_2| = \mathcal{O}(L_1)$, and we assume $|d_1 - d_2| < L_1$ by ignoring the constant factor. Therefore, for a randomly sampled pair (x_0, y_0) that encounters $(\bar{x}, \bar{y})$, we are able to derive a pair of (i, j) such that $d_1 + i \cdot L_1 = d_2 + j \cdot L_2$, as long as their distance bias $d_1 - d_2$ is in the set $\mathcal{D}$. In other words, we are able to *correct such a distance bias by using multi-cycles*. Hereafter, the set $\mathcal{D}$ is referred to as the set of *correctable distance bias*. Thus, the probability of reaching $(\bar{x}, \bar{y})$ from a random pair (x_0, y_0) at a common distance is amplified by roughly t times, where t is the number of cycles to the maximum.

3 Preimage Attack on XOR Combiners Based on the Interchange Structure

In this section, we introduce our first attack — the preimage attack on the XOR combiner. In this attack, we are given an n-bit target value V, and our goal is to find a message $\boldsymbol{M}$ such that $\mathcal{H}_1(\boldsymbol{M}) \oplus \mathcal{H}_2(\boldsymbol{M}) = V$. Notice that, if the goal is to find two messages $\boldsymbol{M}_1$ and $\boldsymbol{M}_2$ such that $\mathcal{H}_1(\boldsymbol{M}_1) \oplus \mathcal{H}_2(\boldsymbol{M}_2) = V$,

[16] This is a very low probability that the set contains repeated values, particularly when t is significantly small compared with L_1. Here, we omit the discussion.

we can immediately launch a meet-in-the-middle procedure to find a solution of the equation $\mathcal{H}_1(\boldsymbol{M}_1) = \mathcal{H}_2(\boldsymbol{M}_2) \oplus V$ with $2^{n/2}$ computations. That is because in the last equation, the left-hand side and right-hand side are independent. By separately computing $2^{n/2}$ values on each side, we obtain 2^n pairs and will find a match with high probability due to the birthday paradox. Thus, the above is essentially to find a collision, which is an easy challenge. However, in the real challenge, the collision must be generated from the same message. Thus, the computations on the two side of the equation are pairwise related, *i.e.*, the computation on one side of the equation can only pair with a single computation on the other side. Consequently, unlike in the easy challenge, $2^{n/2}$ computations on each side can only generate $2^{n/2}$ pairs instead of 2^n. Therefore, to launch a similar meet-in-the-middle procedure as we did in the easy challenge for the real challenge, a crucial part of our attack is to construct a structure breaking the pairwise dependency between the two computations. That structure playing the important role is the *interchange structure* introduced in Sect. 2.5. Next, we provide an overview of our attack based on the interchange structure and then give detailed attack procedures.

3.1 Overview of the Attack

Next, we give an overview of the first preimage attack on the XOR combiner. Let V denote the target value. The two hash functions $\mathcal{H}_1$ and $\mathcal{H}_2$ share the same input message, and hence the internal states of their iterative compression function computations are pairwise related. We first manage to simultaneously control the computation chains of $\mathcal{H}_1$ and $\mathcal{H}_2$ by constructing an interchange structure including a message structure $\mathcal{M}$ and two sets of internal states $\mathcal{A}$ (for $\mathcal{H}_1$) and $\mathcal{B}$ (for $\mathcal{H}_2$) such that for any state A picked from $\mathcal{A}$ and B picked from $\mathcal{B}$, we can easily derive a message $M_{A,B}$ from $\mathcal{M}$ such that $\mathcal{H}_1(M_{A,B})$ produces A and $\mathcal{H}_2(M_{A,B})$ produces B. Hence, we can select states from $\mathcal{A}$ and $\mathcal{B}$ independently in the next phase of the attack. In the next phase, we use a birthday match to find a message block m, a state A in $\mathcal{A}$ and a state B in $\mathcal{B}$ such that $h_1(A, m) \oplus h_2(B, m)$ equals the target hash digest V, where h_1 and h_2 are the compression functions of $\mathcal{H}_1$ and $\mathcal{H}_2$ respectively. Finally, given states A and B, we derive the message $M_{A,B}$ from $\mathcal{M}$, and output $M_{A,B} \parallel m$ as a preimage of V.[17] The birthday match in the second phase of the attack is essentially a meet-in-the-middle procedure enabled by the interchange structure built in the first phase of the attack. Thus, the entire attack is more efficient than a brute-force attack.

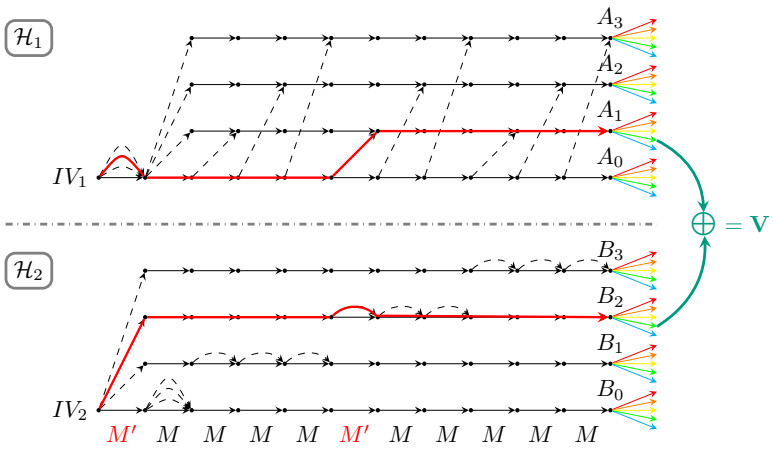
Attack 1: Preimage attack on XOR combiner based on the interchange structure

[17] Note that for simplicity of description, we omit the description of the finalization transformation on the internal state with the padding block and refer to Sect. 3.2 for the formal description.

– **Phase 1:** Build a 2^t-interchange structure using 2^{2t} switches that enables to jump between chains (see. Sect.2.5). This structure has starting points IV_1 and IV_2 and ending points $\{A_j \mid j = 0 \dots 2^t - 1\}$ and $\{B_k \mid k = 0 \dots 2^t - 1\}$, so that for any state pair (A_j, B_k), we can easily select a message ending in it.
– **Phase 2:** Select a random message block m, and compute two lists by evaluating the compression functions after the interchange structure: $\{A'_j = h_1(A_j, m) \mid j = 0 \dots 2^t - 1\}$ and $\{B'_k = V \oplus h_2(B_k, m) \mid k = 0 \dots 2^t - 1\}$. We expect a match between the lists with probability 2^{2t-n}. After about 2^{n-2t} random choices of m, we obtain a match (j^*, k^*):

$$h_1(A_{j^*}, m) = V \oplus h_2(B_{k^*}, m) \quad i.e. \quad h_1(A_{j^*}, m) \oplus h_2(B_{k^*}, m) = V.$$

Therefore, we can construct a preimage of V by concatenating the message leading to (A_{j^*}, B_{k^*}) in the interchange structure and m (we ignore the finalization function here).



The complexity of the preimage search is approximately 2^{n-t} evaluations of the compression function, using an interchange structure with 2^t endpoints.

Complexity Analysis. Building the interchange structures requires approximately $2^{2t+n/2}$ evaluations of the compression function, while the preimage search requires approximately 2^{n-t}. The optimal complexity[18] is reached when both steps take the same time, *i.e.*, $t = n/6$. This gives a complexity of $\tilde{O}(2^{5n/6})$. Since it uses messages of length at least $n/2 \cdot 2^{2t}$, the optimal complexity is obtained for messages of length at least $2^{n/3}$. For messages shorter than $2^{n/3}$, it provides a trade-off of $2^n \cdot L^{-1/2}$ between the maximal allowed message length L and the time complexity of attack (see Fig. 12 for a trade-off curve).

[18] From now on, we will use "optimal complexity" to mean the minimized complexity under the optimal choice of parameters for each attack.

3.2 Details of the Preimage Attack on XOR Combiners Using the Interchange Structure

Now, we describe the full preimage attack in detail. We first build an interchange structure with 2^t chains for each of $\mathcal{H}_1$ and $\mathcal{H}_2$. We denote the ending points as $\{A_j \mid j = 0 \dots 2^t - 1\}$ and $\{B_k \mid k = 0 \dots 2^t - 1\}$, and we know how to select a message $\boldsymbol{M}_{jk}$ to reach any state (A_j, B_k). When adding message fragment $m \parallel pad$, to one of the messages $\boldsymbol{M}_{jk}$ in the interchange structure, where m is a message block and pad is the final block padded with the length L of the preimage message, the output of the combiner can be written as follows:

$$\mathcal{H}_1(\boldsymbol{M}_{jk} \| m \| pad) \oplus \mathcal{H}_2(\boldsymbol{M}_{jk} \| m \| pad) = h_1(h_1(A_j, m), pad) \oplus h_2(h_2(B_k, m), pad),$$

Note that we fix the finalization functions of $\mathcal{H}_1$ and $\mathcal{H}_2$ as their compression functions, h_1 and h_2, respectively.

To reach a target value V, we select a random block m, and we evaluate $\{A'_j = h_1(h_1(A_j, m), pad) \mid j = 0 \dots 2^t - 1\}$ and $\{B'_k = V \oplus h_2(h_2(B_k, m), pad) \mid k = 0 \dots 2^t - 1\}$. If there is a match (j^*, k^*) between the two lists, we have the following:

$$\begin{aligned} A'_{j^*} = B'_{k^*} &\Leftrightarrow h_1(h_1(A_{j^*}, m), pad) = V \oplus h_2(h_2(B_{k^*}, m), pad) \\ &\Leftrightarrow \mathcal{H}_1(\boldsymbol{M}_{jk} \parallel m \parallel pad) \oplus \mathcal{H}_2(\boldsymbol{M}_{jk} \parallel m \parallel pad) = V. \end{aligned}$$

For a random choice of m, we expect that a match exists with probability 2^{2t-n}, and testing it requires approximately 2^t operations[19]. We will have to repeat this procedure 2^{n-2t} times on average; therefore, the total cost of the preimage search is approximately 2^{n-t} evaluations of h_1 and h_2.

As explained in the previous section, building a 2^t-interchange structure requires approximately $n/2 \cdot 2^{2t+n/2}$ operations. Using $t = n/6$, we balance the two steps of the attack and reach the optimal complexity of approximately $n/2 \cdot 2^{5n/6}$ operations for this preimage attack.

4 Improved Preimage Attack on XOR Combiners Based on Deep Iterates

The first attack works identically for the case in which the combined hash functions use the HAIFA mode, and the case in which they use the MD construction. However, when they are limited to use the MD construction, we can launch a more efficient attack than the first. In this case, pairwise dependency between internal states can be broken efficiently by using repeated message blocks. Explicitly, we use a different approach to get the two sets of states $\mathcal{A} = \{A_j \mid j = 0 \dots 2^t - 1\}$ and $\mathcal{B} = \{B_k \mid k = 0 \dots 2^t - 1\}$ such that for any pair of states $(A_j, B_k \mid A_j \in \mathcal{A}, B_k \in \mathcal{B})$, we can manage to find a message $M_{A,B}$ such that $(IV_1, IV_2) \xrightarrow{M_{A,B}} (A_j, B_k)$. For convenience, we name such an

[19] It takes $O(t \cdot 2^t)$ operations by sorting the lists, but only $2 \cdot 2^t$ using a hash table.

abstract procedure `GenPairableStates`, which is implemented and utilized by quite different approaches in different attacks – the first attack implements it using interchange structure, this second attack implements it using deep iterates in functional graphs, and as will be seen, the third attack implements it using cyclic nodes in functional graphs.

The first step is to fix an arbitrary message block m to the compression functions, giving rise to n to n-bit random mappings $f_1(\cdot) \triangleq h_1(\cdot, m)$ and $f_2(\cdot) \triangleq h_2(\cdot, m)$. Such random mappings and their functional graphs have many interesting properties and have been extensively studied and used in cryptanalysis, as shown in Sect. 2.7. However, to attack hash combiners, we exploit them in new ways. In this attack, instead of precisely controlling every computational step in chains of equal length to obtain two sets of endpoints as in building an interchange structure, here, we loosely herd computational chains of various length to collect two sets of states. These collected states have large offsets in the chains. These chains are iteratively computed using the above defined random mappings f_1 and f_2. Thus, the collected states are essentially *deep iterates* in the functional graphs of f_1 and f_2, which are introduced in Sect. 2.7. As has been shown in Sect. 2.7, such special states are relatively easy (*i.e.*, with high probability) to be reached from randomly selected starting states. This is where the advantage of the attack mainly comes from.

In this attack, given a pair of such special states (A_j, B_k), finding a common message mapping a pair of starting states to them under the two hash computations is not as efficient as selecting a message from an interchange structure in Attack 1. However, collecting those final states by expanding functional graphs is much more efficient than computing endpoints by building an interchange structure. This attack amortizes computational costs to different steps. Thus, it provides better balance between different attack steps. Moreover, it also provides a better trade-off between the message length and time complexity.

However, unlike the interchange-structure-based attack, this approach uses chains of various lengths, which implies that lengths of message fragments in intermediate attack steps are not fixed in advance. However, the length of the preimage needs to be predefined in this attack. Thus, the length padding at the end of the hash computations will be a problem. We overcome this problem using our tool, the simultaneous expandable message for two MD hash functions, which is introduced in Sect. 2.6.

Next, we provide a high-level overview of this attack and then present the detailed attack steps.

4.1 Overview of the Attack

Suppose that we are given a target n-bit preimage value V and our goal is to find a message $\boldsymbol{M}$ such that $\mathcal{H}_1(\boldsymbol{M}) \oplus \mathcal{H}_2(\boldsymbol{M}) = V$. Although the formal problem does not restrict $\boldsymbol{M}$ in any way, several concrete hash functions restrict the length of $\boldsymbol{M}$. Therefore, we will first assume that the size of $\boldsymbol{M}$ is bounded by a parameter L.

The attack is composed of three main phases.

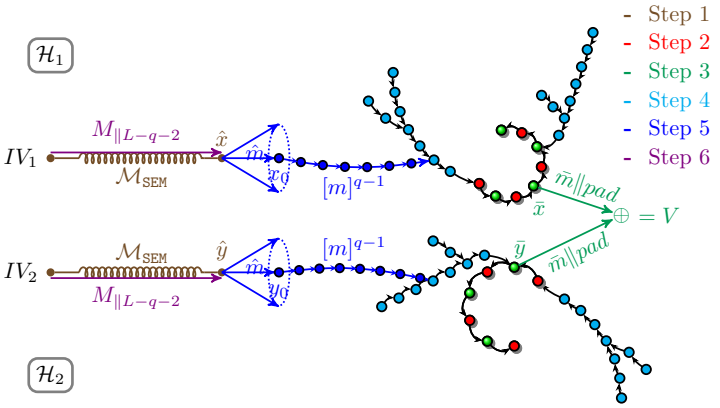
Attack 2: Preimage attack on XOR combiner based on deep iterates

- **Phase 1:** Build a simultaneously expandable message $\mathcal{M}_{\texttt{SEM}}$ for $\mathcal{H}_1$ and $\mathcal{H}_2$, starting from the initial state pair (IV_1, IV_2) and ending with state pair $(\hat{x}, \hat{y})$, such that for each length κ in some appropriate range (which is roughly $[n^2, L]$), there is a message $M_{\|\kappa}$ of κ blocks that maps (IV_1, IV_2) to $(\hat{x}, \hat{y})$.
- **Phase 2:** Find a set $\mathcal{S}$ (of size 2^s) of tuples of the form $((x, y), w)$ such that w is a single block, $(x, y) \xrightarrow{w} (a, b)$, and $h_1(a, pad) \oplus h_2(b, pad) = V$, where pad is the final block of the (padded) preimage message of length L. Moreover, (x, y) has a special property that will be defined in the detailed description of this phase.
- **Phase 3:** Start from $(\hat{x}, \hat{y})$ and compute a message fragment $\hat{M}_{\|q}$ of length q (shorter than $L - 2$) such that $(\hat{x}, \hat{y}) \xrightarrow{\hat{M}_{\|q}} (\bar{x}, \bar{y})$ for some $((\bar{x}, \bar{y}), \bar{m}) \in \mathcal{S}$. For this tuple, denote $(\bar{a}, \bar{b}) \triangleq h_{1,2}((\bar{x}, \bar{y}), \bar{m})$.

Finally, we pick a message of $L - q - 2$ blocks from $\mathcal{M}_{\texttt{SEM}}$, denoted by $M_{\|L-q-2}$, giving $(IV_0, IV_1) \xrightarrow{M_{\|L-q-2}} (\hat{x}, \hat{y})$, and concatenate $M_{\|L-q-2} \,\|\, \hat{M}_{\|q} \,\|\, \bar{m}$ in order to reach the state pair $(\bar{a}, \bar{b})$ from (IV_1, IV_2) with a message of length $L - 1$. Indeed, we have

$$(IV_0, IV_1) \xrightarrow{M_{\|L-q-2}} (\hat{x}, \hat{y}) \xrightarrow{\hat{M}_{\|q}} (\bar{x}, \bar{y}) \xrightarrow{\bar{m}} (\bar{a}, \bar{b}).$$

Altogether, we obtain the padded preimage for the XOR combiner: $\boldsymbol{M} = M_{\|L-q-2} \,\|\, \hat{M}_{\|q} \,\|\, \bar{m} \,\|\, pad$.



Complexity Analysis. Denote $L = 2^\ell$. For parameters $g_1 \geq max(n/2, n - \ell)$ and $s \geq 0$, the complexity of phases of the attack (as computed in their detail

description) is given below (ignoring constant factors).

Phase 1: $2^{\ell} + n^2 \cdot 2^{n/2}$ **Phase 2:** 2^{n+s-g_1}

Phase 3: $2^{3g_1/2-s/2} + L \cdot 2^{9g_1/2-2n-3s/2} + L \cdot 2^{2g_1-n}$
$= 2^{3g_1/2-s/2} + 2^{\ell+9g_1/2-2n-3s/2} + 2^{\ell+2g_1-n}$

We balance the time complexities of the second phase and the first term in the expression of the third phase by setting $n + s - g_1 = 3g_1/2 - s/2$, or $s = 5g_1/3 - 2n/3$, giving a value of $2^{n/3+2g_1/3}$ for these terms. Furthermore, $\ell + 9g_1/2 - 2n - 3s/2 = \ell + 2g_1 - n$, and the time complexity expression of Phase 3 is simplified to be $2^{n/3+2g_1/3} + 2^{\ell+2g_1-n}$. Since g_1 is a positive factor in all the terms, we optimize the attack by picking the minimal value of g_1 under the restriction $g_1 \geq \max(n/2, n-\ell)$. In case $\ell \leq n/2$, we set $g_1 = n - \ell$ and the total time complexity of the attack[20] is

$$2^{n/3+2(n-\ell)/3} = 2^{n-2\ell/3}.$$

The optimal complexity is $2^{2n/3}$, obtained for messages of length $2^{n/2}$ (see Fig. 12 for a trade-off curve).

4.2 Details of the Preimage Attack on XOR Combiners using Deep Iterates

Details of Phase 1 can be found in Sect.2.6. In the following, we describe details of the other two phases.

Details of Phase 2: Finding a Set of Target State Pairs. In the second phase, we fix an arbitrary message block m, giving rise to the functional graphs $\mathcal{FG}_{f_1}$ and $\mathcal{FG}_{f_2}$ defined by the random mappings $f_1(\cdot) \triangleq h_1(\cdot, m)$ and $f_2(\cdot) \triangleq h_2(\cdot, m)$. Given parameters $g_1 \geq n/2$ and $s \geq 0$, our goal is to compute a set $\mathcal{S}$ (of size 2^s) of tuples of the form $((x, y), w)$, where w is a single block such that for each tuple, the following hold:

1. The state x is a 2^{n-g_1}-th iterate in $\mathcal{FG}_{f_1}$, and y is a 2^{n-g_1}-th iterate in $\mathcal{FG}_{f_2}$.
2. $(x, y) \xrightarrow{w} (a, b)$ and $h_1(a, pad) \oplus h_2(b, pad) = V$, where pad is a final block of the (padded) preimage message of length L.

This algorithm resembles the algorithm used in the final phase in previous interchange-structure-based preimage attack (Attack 1 in Sect. 3), as both look for state pairs (x, y) that give $h_1(x, w \parallel pad) \oplus h_2(y, w \parallel pad) = V$ (for some message block w). The difference is that in previous interchange-structure-based attack, (x, y) is an arbitrary endpoint pair of the interchange structure, while in this case, we look for x and y that are deep iterates.

[20] Note that $\ell + 2g_1 - n = n - \ell < n - 2\ell/3$.

Phase 2 of Attack 2: Finding a set of target state pairs

1. Fix an arbitrary single-block value m, and construct $f_1(\cdot) \triangleq h_1(\cdot, m)$ and $f_2(\cdot) \triangleq h_2(\cdot, m)$.
2. Expand $\mathcal{FG}_{f_1}$ using Alg. 5 with parameter g_1. Store all encountered 2^{n-g_1}-th iterates in a table $\mathcal{T}_1$.
3. Expand $\mathcal{FG}_{f_2}$ using Alg. 5 with parameter g_1. Store all encountered 2^{n-g_1}-th iterates in a table $\mathcal{T}_2$.
4. Allocate a set $\mathcal{S} = \emptyset$. For single-block values $w = 0, 1, \ldots$, perform the following steps until $\mathcal{S}$ contains 2^s elements:
   (a) For each node $x \in \mathcal{T}_1$ evaluate $h_1(x, w \,\|\, pad)$, and store the results in a table $\mathcal{T}_1'$, sorted according $h_1(x, w \,\|\, pad)$.
   (b) For each node $y \in \mathcal{T}_2$ evaluate $h_2(y, w \| pad) \oplus V$, and look for matches $h_2(y, w \,\|\, pad) \oplus V = h_1(x, w \,\|\, pad)$ with $\mathcal{T}_1'$. For each match, add the tuple $((x, y), w)$ to $\mathcal{S}$.

The time complexity of steps 2 and 3 is approximately 2^{g_1}. The time complexity of step 4.(a) and step 4.(b) is also bounded by 2^{g_1}. We now calculate the expected number of executions of Step 4 until 2^s matches are found and inserted into $\mathcal{S}$.

According to Observation 1 in Sect. 2.7, the expected size of $\mathcal{T}_1$ and $\mathcal{T}_2$ (the number of deep iterates) is close to 2^{g_1}. Thus, for each execution of Step 4, the expected number of matches on n-bit values $h_2(y, w \,\|\, pad) \oplus V = h_1(x, w \,\|\, pad)$ is 2^{2g_1-n}. Consequently, Step 4 is executed 2^{s+n-2g_1} times in order to obtain 2^s matches. Altogether, the total time complexity of this step is

$$2^{n+s-2g_1+g_1} = 2^{n+s-g_1}.$$

Details of Phase 3: Hitting a Target State Pair. In the third and final phase, we start from $(\hat{x}, \hat{y})$ and compute a message $\hat{M}_{\|q}$ of length q (is valid as long as shorter than $L-2$) such that $(\hat{x}, \hat{y}) \xrightarrow{\hat{M}_{\|q}} (\bar{x}, \bar{y})$ for some $((\bar{x}, \bar{y}), \bar{m}) \in \mathcal{S}$. We use in a strong way the fact that states $\bar{x}$ (and $\bar{y}$) in $\mathcal{S}$ are deep iterate (of depth 2^{n-g_1}) in $\mathcal{FG}_{f_1}$ (and $\mathcal{FG}_{f_2}$).

The goal of this phase is to find a pair of starting points of chains, reaching some $((\bar{x}, \bar{y}), \bar{m}) \in \mathcal{S}$ at same distance. This phase is carried out by picking an arbitrary starting message block $\hat{m}$, which gives points $x_0 = h_1(\hat{x}, \hat{m})$ and $y_0 = h_2(\hat{y}, \hat{m})$. We then continue to evaluate the chains $x_{i+1} = h_1(x_i, m)$ and $y_{j+1} = h_2(y_j, m)$ up to length at most $L-3$. We hope to encounter $\bar{x}$ at some distance $q-1$ from x_0 and to encounter $\bar{y}$ at the same distance $q-1$ from y_0, where $((\bar{x}, \bar{y}), \bar{m}) \in \mathcal{S}$. In case in which for all pairs of $((\bar{x}, \bar{y}), \bar{m}) \in \mathcal{S}$, $\bar{x}$ and $\bar{y}$ are encountered at different distances in the chains, or at least one of them is not encountered at all, we pick a different value for $\hat{m}$ and start again. Once we find such a value for $\hat{m}$ and pair of iterates $(\bar{x}, \bar{y})$, this gives the required $\hat{M}_{\|q} \triangleq \hat{m} \,\|\, [m]^{q-1}$.

According to Observation 2 in Sect. 2.7, for a pair of 2^{n-g_1}-th iterates $\bar{x}$ and $\bar{y}$, the probability that they will be encountered at the same distance from arbitrary starting points x_0 and y_0 of chains is $(2^{n-g_1})^3 \cdot 2^{-2n} = 2^{n-3g_1}$. Since $\mathcal{S}$ contains 2^s elements, the probability calculation yields the conclusion that we need to compute about 2^{3g_1-n-s} chains from different starting points to find such a value for $\hat{m}$ generating starting points (x_0, y_0) reaching a pair of deep iterates $(\bar{x}, \bar{y})$ in $\mathcal{S}$ at the same distance.

The next question which we address is to what maximal length L' should we evaluate chains $\vec{x}$ and $\vec{y}$. As we wish to reach iterates $\bar{x}$ and $\bar{y}$ of depth 2^{n-g_1}, it can be shown that $L' = 2^{n-g_1}$ is optimal. Since the total chain length should be less than $L-3$, this impose the restriction $L' = 2^{n-g_1} < L-3$, or $2^{g_1} < 2^n/L$.

The naive algorithm described above performs about 2^{3g_1-n-s} trials, where each trial evaluates chains of length $L' = 2^{n-g_1}$ from arbitrary points, giving a total time complexity of approximately $2^{3g_1-n-s+n-g_1} = 2^{2g_1-s}$. Since $g_1 \geq n/2$, the time complexity of this phase is at least 2^{n-s}, and after making a balance with that of the Phase 2, the time complexity can be $2^{3n/4}$ by setting $s = n/4$.

However, it is possible to optimize this naive algorithm by further expanding the graphs of f_1 and f_2. As a result, the evaluated chains are expected to collide with the graphs sooner (before they are evaluated to the full length of 2^{n-g_1}). Once a collision occurs, we use a look-ahead procedure to calculate the distance of the chain's starting point from $\bar{x}$ (or $\bar{y}$) in each tuple $((x, y), w) \in \mathcal{S}$. This look-ahead procedure resembles the one used in attacks on hash-based MACs [GPSW14,PW14] (although the setting and actual algorithm in our case are obviously different).

We define an $\mathcal{S}$-*node* (for f_1) as a node x such that there exists a node y and a message block w such that $((x, y), w) \in \mathcal{S}$. An $\mathcal{S}$-node for f_2 is defined in a similar way. To avoid heavy update operations for the distances from all the $\mathcal{S}$-nodes, we use distinguished points. Essentially, each computed chain is partitioned into intervals according to distinguished points, where each distinguished point stores only the distances to all the $\mathcal{S}$-nodes that are contained in its interval up to the next distinguished point. Given a parameter $g_2 > g_1$, the algorithm for this phase is described below.

Phase 3 of Attack 2: Hitting a target state pair

1. Develop (about) 2^{g_2} nodes in $\mathcal{FG}_{f_1}$ (and $\mathcal{FG}_{f_2}$) (as specified in Alg. 5) with the following modifications.
   - Store only distinguished points for which the $n - g_2$ LSBs are zero.
   - Once an $\mathcal{S}$-node is encountered, update its distance in the previously encountered distinguished point (which is defined with high probability[a]).
   - Stop evaluating each chain once it hits a stored distinguished point.
   - The evaluated distinguished points for f_1 (resp. f_2) are stored in the data structure $\mathcal{G}_1$ (resp. $\mathcal{G}_2$).

2. For single-block values $\hat{m} = 0, 1, \ldots$, compute $x_0 = h_1(\hat{x}, \hat{m})$ and $y_0 = h_2(\hat{y}, \hat{m})$ and repeat the following step:
   (a) Compute chains $\vec{x}$ and $\vec{y}$ as specified below.
   - First, compute the chains in a standard way by evaluating the compression functions h_1 and h_2 until they hit stored distinguished points in $\mathcal{G}_1$ and $\mathcal{G}_2$ (respectively).
   - Then, allocate a table $\mathcal{T}_1$ for f_1 (and $\mathcal{T}_2$ for f_2) and continue traversing (only) the distinguished points of the chain (using the links in $\mathcal{G}_1$ and $\mathcal{G}_2$) up to depth $L-2$, while updating $\mathcal{T}_1$ (resp. $\mathcal{T}_2$): for each visited distinguished point, add all its stored $\mathcal{S}$-nodes to $\mathcal{T}_1$ (resp. $\mathcal{T}_2$) with its distance from x_0 (resp. y_0).
   - Once the maximal depth $L-2$ is reached, sort $\mathcal{T}_1$ and $\mathcal{T}_2$. Search for nodes $\bar{x}$ and $\bar{y}$ that were encountered at the same distance $q-1$ from x_0 and y_0 (respectively), such that $((\bar{x}, \bar{y}), \bar{m}) \in \mathcal{S}$. If such $\bar{x} \in \mathcal{T}_1$ and $\bar{y} \in \mathcal{T}_2$ exist, return the message $\hat{M}_{\|q} = \hat{m} \parallel [m]^{q-1}$ and $\bar{m}$ (retrieved from $\mathcal{S}$) as output. Otherwise (no such $\bar{x}$ and $\bar{y}$ were found), return to Step 2.

[a] Since $g_2 > g_1$, $\mathcal{S}$-nodes are deeper iterates than distinguished points, and thus distinguished points are likely to be encountered in an arbitrary chain before an $\mathcal{S}$-node.

The time complexity of Step 1 is approximately 2^{g_2} (note that we always perform a constant amount of work per developed node).

For time complexity of Step 2, the analysis is as follows. As concluded above, the expected number of values for $\hat{m}$ we need to test until we find a pair of starting point (x_0, y_0) of chains encounter at the same distance to a pair of 2^{n-g_1}-th iterates $(\bar{x}, \bar{y})$ in $\mathcal{S}$ is approximately 2^{3g_1-n-s}.

The analysis of the complexity of Step 2.(a) is as follows. First, we estimate the expected number of nodes that we visit during the computation of a chain. Initially, we compute approximately 2^{n-g_2} nodes until we hit stored distinguished points. Then, we continue by traversing (only) distinguished points up to depth of about L. The expected number of such points is $L \cdot 2^{g_2-n}$. Therefore, we expect to visit approximately $2^{n-g_2} + L \cdot 2^{g_2-n}$ nodes while computing a chain. Finally, we need to account for all the $\mathcal{S}$-nodes encountered while traversing the chains of depth L. Basically, there are 2^s $\mathcal{S}$-nodes which are iterates of depth 2^{n-g_1}, (essentially) randomly chosen in Phase 2 out of approximately 2^{g_1} such deep iterates. As a result, the probability of such a deep iterate to be a $\mathcal{S}$-node is approximately 2^{s-g_1} (while other nodes have probability 0). Therefore, while traversing chains of depth L, we expect to encounter at most $L \cdot 2^{s-g_1}$ $\mathcal{S}$-nodes (which is a bound on the sizes of $\mathcal{T}_1$ and $\mathcal{T}_2$). Altogether, the expected time complexity of a single execution of Step 2.(a) is at most $2^{n-g_2} + L \cdot 2^{g_2-n} + L \cdot 2^{s-g_1}$.

The total time complexity of this phase is $2^{g_2} + 2^{3g_1-n-s} \cdot (2^{n-g_2} + L \cdot 2^{g_2-n} + L \cdot 2^{s-g_1}) = 2^{g_2} + 2^{3g_1-g_2-s} + L \cdot 2^{3g_1+g_2-2n-s} + L \cdot 2^{2g_1-n}$. We set $g_2 = 3g_1/2 - s/2$

which balances the first two terms and gives a time complexity of

$$2^{3g_1/2-s/2} + L \cdot 2^{9g_1/2-2n-3s/2} + L \cdot 2^{2g_1-n}.$$

The time complexity evaluation of the full attack at the beginning of this section shows that for the optimal parameters of this attack, the extra two terms $L \cdot 2^{9g_1/2-2n-3s/2} + L \cdot 2^{2g_1-n}$ are negligible compared to the other terms in the complexity equation. In other words, the distinguished points method allowed us to resolve with no overhead the complication of keeping track of distances from the $\mathcal{S}$-nodes.

4.3 Optimizing the Deep-Iterates-Based Preimage Attack on XOR Combiners using the Interchange Structure

The above deep-iterates-based preimage attack on XOR combiners can be slightly improved using an interchange structure. Recall that the interchange structure helps to break the dependency between two hash computations on a common message. When building a 2^r-interchange structure starting from the pair of endpoints $(\hat{x}, \hat{y})$ of the simultaneous expandable message and ending at two sets of states $\mathcal{A} = \{A_1, A_2, \ldots, A_{2^r}\}$ and $\mathcal{B} = \{B_1, B_2, \ldots, B_{2^r}\}$, any $A_i \in \mathcal{A}$ can make a pair with any $B_j \in \mathcal{B}$ (for any such a pair, one could easily find a common message mapping $(\hat{x}, \hat{y})$ to this pair). Therefore, by using a single message block $\hat{m}$ to generate two sets of 2^r random starting nodes respectively from $\mathcal{A}$ and $\mathcal{B}$, we can get 2^{2r} pairs of starting nodes. As a result, the required number of samplings on the random massage block $\hat{m}$ is reduced by a factor of 2^{2r}.

The detailed complexity analysis of the attack using a 2^r-interchange structure is as follows: Denote $L = 2^\ell$. For parameters $g_1 \geq \max(n/2, n-\ell)$, $g_2 \geq 0$, $s \geq 0$ and $0 \leq r \leq \ell/2$ (because the length 2^{2r} of the interchange structure should be less than the message length 2^ℓ), the complexity of phases of the attack is given below (ignoring constant factors).

Phase 1: $2^\ell + n^2 \cdot 2^{n/2}$ **Phase 2:** 2^{n+s-g_1}

Phase 3: $2^{g_2} + 2^{3g_1-n-s-2r} \cdot 2^r \cdot (2^{n-g_2} + 2^{\ell+g_2-n} + 2^{\ell+s-g_1}) + 2^{n/2+2r}$

$$= 2^{g_2} + 2^{3g_1-g_2-s-r} + 2^{3g_1+g_2+\ell-2n-s-r} + 2^{2g_1+\ell-n-r} + 2^{n/2+2r}$$

Compared with the complexity of the attack in Sect.4.1, the difference lies in Phase 3. In the complexity formula of Phase 3, the term 2^{g_2} is the number of nodes developed in the look-ahead procedure; the term $2^{3g_1-n-s-2r}$ is the required number of samplings on the value of message block $\hat{m}$ to get pairs of starting nodes, which is reduced by a factor of 2^{2r} when building a 2^r-interchange structure; the term $2^r \cdot (2^{n-g_2} + 2^{\ell+g_2-n} + 2^{\ell+s-g_1})$ is the time complexity for computing distances of pairs of starting nodes (generated using the same value for $\hat{m}$) from all 2^s target nodes; the term $2^{n/2+2r}$ is the time complexity for building the 2^r-interchange structure.

We first balance the first two terms in Phase 3 by setting $g_2 = 3g_1 - g_2 - s - r$, which gives $g_2 = 3g_1/2 - s/2 - r/2$. Thus, the complexity of Phase 3

becomes $2^{3g_1/2-s/2-r/2}+2^{9g_1/2-3s/2-3r/2+\ell-2n}+2^{2g_1+\ell-n-r}+2^{n/2+2r}$. We then balance Phase 2 and Phase 3 by setting $n+s-g_1 = 3g_1/2-s/2-r/2$, which gives $s = 5g_1/3-r/3-2n/3$. The sum of all dominant terms turns to be $2^{\ell}+2^{n/3+2g_1/3-r/3}+2^{2g_1+\ell-n-r}+2^{n/2+2r}$. Finally, we pick the minimal value of g_1 under the restriction $g_1 \geq \max(n/2, n-\ell)$. In case $\ell \leq n/2$, we set $g_1 = n-\ell$. The sum of dominant terms turns to be $2^{\ell}+2^{n-2\ell/3-r/3}+2^{n-\ell-r}+2^{n/2+2r}$. Considering $n-2\ell/3-r/3 > n-\ell-r$ always holds, the sum of dominant terms is

$$2^{n-2\ell/3-r/3}+2^{n/2+2r}.$$

Note that there is a restriction on r, that is, $r \leq \ell/2$. As a result (see Fig. 12 for a trade-off curve),

- For the case $\ell \leq 3n/11$, we have $n-2\ell/3-r/3 > n/2+\ell > n/2+2r$. We set $r = \ell/2$ to optimize the complexity. Thus, the sum of dominant terms is $2^{n-5\ell/6}$. The optimal complexity is $2^{17n/22}$ obtained for message of length $2^{\ell} = 2^{3n/11}$.
- For the case $3n/11 < \ell \leq n/2$, we set $r = 3n/14-2\ell/7$ to make a balance, which fulfils $r < \ell/2$ in this case. The sum of dominant terms is $2^{13n/14-4\ell/7}$. The optimal complexity is $2^{9n/14}$ obtained for message of length $2^{\ell} = 2^{n/2}$.

5 Improved Preimage Attack on XOR Combiners Based on Multi-Cycles

When the underlying hash functions use the MD construction, and the maximum length of the message is allowed to exceed $2^{n/2}$ blocks, we can further improve previous deep-iterates-based preimage attack. The idea is that we utilize more special nodes in function graphs, which are called cyclic nodes, and exploit a technique named *multi-cycles* as introduced in Sect. 2.7. Recall that, in the deep-iterates-based attack, a key step is to find two starting nodes x_0 and y_0 in functional graphs of f_1 and f_2, such that they reach the selected target nodes $\bar{x}$ and $\bar{y}$ at a common distance. We find that when selecting cyclic nodes as target nodes $\bar{x}$ and $\bar{y}$, the probability of a pair of random node (x_0, y_0) reaching them at a common distance can be greatly amplified. Indeed, cyclic nodes are essentially special deep iterates that are located not only *deep* in the functional graph but also in a *cycle* of the graph. Therefore, for two cyclic nodes in two independent functional graphs, by looping around the cycles, some differences between distances from two random nodes to the two cyclic nodes can be corrected by the difference between the two cycle lengths. More precisely, if the members of a target node pair $(\bar{x}, \bar{y})$ are both cyclic nodes within the largest components in two functional graphs, the probability of a random pair (x_0, y_0) reaching $(\bar{x}, \bar{y})$ at a common distance is amplified by $\#C$ times, the maximum number of cycles that can be used, by using the set of correctable distance bias as stated in Sect. 2.7. Moreover, such a probability amplification comes with almost no increase of complexity at Step 2, which leads to a new complexity trade-off between Steps 2 and 3. Thus, the usage of cyclic nodes and multi-cycles enables

us to reduce the computational complexity of preimage attacks on the XOR combiner.

5.1 Overview of the Attack

Here, we briefly list the *main* steps of our preimage attack on the XOR combiner.

Attack 3: Preimage attack on the XOR combiner based on multi-cycles

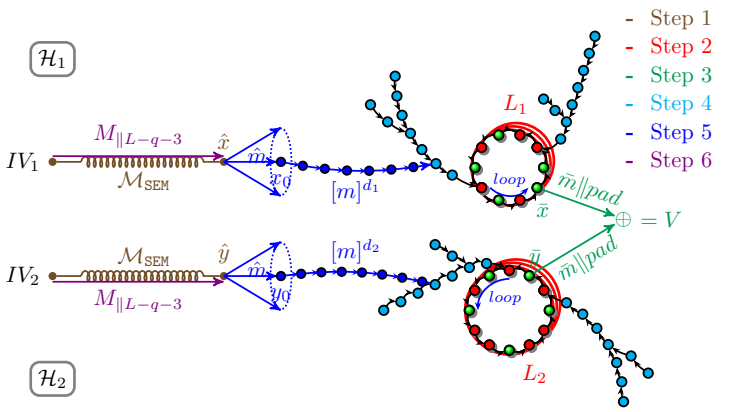
- **Phase 1:** Build a simultaneous expandable message $\mathcal{M}_{\texttt{SEM}}$ for $\mathcal{H}_1$ and $\mathcal{H}_2$, starting from (IV_1, IV_2) and ending with $(\hat{x}, \hat{y})$.
- **Phase 2:** Collect cyclic nodes within the largest components of functional graphs $\mathcal{FG}_{f_1}$ and $\mathcal{FG}_{f_2}$ and compute the set of correctable distance bias
$$\mathcal{D} = \{i \cdot \Delta L \mod L_1 \mid i = 0, 1, \ldots, \#C\},$$
where L_1 and L_2 are the cycle lengths of the largest components of $\mathcal{FG}_{f_1}$ and $\mathcal{FG}_{f_2}$, respectively, and $\Delta L = L_2 - L_1 \mod L_1$.
- **Phase 3:** Find a set $\mathcal{S}$ (of size 2^s) of tuples of the form $((x, y), w)$ such that x and y are cyclic nodes located in the largest components of $\mathcal{FG}_{f_1}$ and $\mathcal{FG}_{f_2}$, respectively, and $(x, y) \xrightarrow{w} (a, b)$ and $h_1(a, pad) \oplus h_2(b, pad) = V$, where pad is the final block of the (padded) preimage message of length L, V is the target hash digest.
- **Phase 4:** Find a message fragment $M_{\texttt{Link}}$ that maps $(\hat{x}, \hat{y})$ to a pair of target nodes $(\bar{x}, \bar{y})$ for some $((\bar{x}, \bar{y}), \bar{m}) \in \mathcal{S}$.
That is done by first start from $(\hat{x}, \hat{y})$, enumerate a message block $\hat{m}$ to find a pair of states (x_0, y_0) with $x_0 = h_1(\hat{x}, \hat{m})$ and $y_0 = h_2(\hat{y}, \hat{m})$, such that in $\mathcal{FG}_{f_1}$ and $\mathcal{FG}_{f_2}$, their distance difference $d_1 - d_2 \mod L_1$ from $(\bar{x}, \bar{y})$ for some $((\bar{x}, \bar{y}), \bar{m}) \in \mathcal{S}$ belongs to $\mathcal{D}$. Suppose the common distance after correcting by the cycle lengths is $q \triangleq d_1 + i \cdot L_1 = d_2 + j \cdot L_2$, and define a message fragment $M_{\texttt{Link}} \triangleq \hat{m} \parallel [m]^q$.

Up to now, we are able to derive a message $M_{\parallel L-q-3}$ from the simultaneous expandable message $\mathcal{M}_{\texttt{SEM}}$ with an appropriate length $L-q-3$ and produce a preimage message:

$$\begin{aligned} \boldsymbol{M} &\triangleq M_{\parallel L-q-3} \parallel M_{\texttt{Link}} \parallel \bar{m} \parallel pad \\ &= M_{\parallel L-q-3} \parallel \hat{m} \parallel [m]^q \parallel \bar{m} \parallel pad. \end{aligned}$$

such that

$$(IV_1, IV_2) \xrightarrow{M_{\parallel L-q-3}} (\hat{x}, \hat{y}) \xrightarrow{\hat{m} \parallel [m]^q} (\bar{x}, \bar{y}) \xrightarrow{\bar{m} \parallel pad} (\mathcal{H}_1(\boldsymbol{M}), \mathcal{H}_2(\boldsymbol{M})) : \mathcal{H}_1(\boldsymbol{M}) \oplus \mathcal{H}_2(\boldsymbol{M}) = V$$



By balancing the complexities of these steps, we obtain an optimal complexity of $2^{5n/8}$.

5.2 Details of the Preimage Attack on the XOR Combiner Using Multi-Cycles

In this section, we present the completed description of the attack procedure and complexity evaluation. We point out the length of our preimage is at least $2^{n/2}$ blocks due to the usage of (multi-) cycles.

Attack Procedure. Denote by V the target hash digest. Suppose the attacker is going to produce a preimage message with a length L. The value of L will be discussed later. The attack procedure is described below.

Detailed Steps of Attack 3

1. Build a simultaneous expandable message structure $\mathcal{M}_{\mathtt{SEM}}$ with a range of length being roughly $[n^2, L]$, starting from the initial state pair (IV_1, IV_2) and ending with a pair of final state $(\hat{x}, \hat{y})$ such that for each positive integer κ of an integer interval, there is a message $M_{\|\kappa}$ with a block length κ in $\mathcal{M}_{\mathtt{SEM}}$ such that $(IV_1, IV_2) \xrightarrow{M_{\|\kappa}} (\hat{x}, \hat{y})$.
2. Fix a single-block message m, and construct $f_1(\cdot) \triangleq h_1(\cdot, m)$ and $f_2(\cdot) \triangleq h_2(\cdot, m)$.
   (a) Run the cycle search algorithm several times to find the cycle length L_1 (resp. L_2) and cyclic nodes within the largest components in $\mathcal{FG}_{f_1}$ (resp. $\mathcal{FG}_{f_2}$). Store all the cyclic nodes in a table $\mathcal{T}_1$ (resp. $\mathcal{T}_2$), denote $\mathcal{T}_1 = \{x_1, x_2, \ldots, x_{L_1}\}$ (resp. $\mathcal{T}_2 = \{y_1, y_2, \ldots, y_{L_2}\}$). Without loss of generality, assume $L_1 \leq L_2$.
   (b) Compute $\#C = \lfloor L/L_1 \rfloor$ as the maximum number of cycles that can be used to correct the distance bias. Compute $\Delta L = L_2 \mod L_1$,

and then compute the set of correctable distance bias:

$$\mathcal{D} = \{i \cdot \Delta L \mod L_1 \mid i = 0, 1, 2, \ldots, \#C\}.$$

3. Find a set

$$\mathcal{S} = \{((\bar{x}_1, \bar{y}_1), \bar{m}_1), ((\bar{x}_2, \bar{y}_2), \bar{m}_2)\}, \cdots, ((\bar{x}_{2^s}, \bar{y}_{2^s}), \bar{m}_{2^s})$$

such that $\bar{x}_i$ and $\bar{y}_i$ are cyclic nodes respectively located in the largest components of $\mathcal{FG}_{f_1}$ and $\mathcal{FG}_{f_2}$, and $(x_i, y_i) \xrightarrow{w} (a_i, b_i)$ and $h_1(a_i, pad) \oplus h_2(b_i, pad) = V$, where pad is the final block of the (padded) preimage message of length L, V is the target hash digest. The search procedure is described as follows.

   (a) Initialize a set $\mathcal{S}$ as empty.
   (b) Select a random single-block message w.
   (c) Compute $h_1^*(x, w \,\|\, pad)$ for each cyclic node x in $\mathcal{T}_1$, and store them in a table $\mathcal{T}_1'$.
   (d) Similarly, for each cyclic node y in $\mathcal{T}_2$, compute $h_2^*(y, w \,\|\, pad) \oplus V$, and look for matches with elements in $\mathcal{T}_1'$. If it is matched to some $h_1^*(x, w \,\|\, pad)$, we have $h_2^*(y, w \,\|\, pad) \oplus h_1^*(x, w \,\|\, pad) = V$, store $((x, y), w)$ in $\mathcal{S}$.
   (e) If $\mathcal{S}$ contains less than 2^s elements, goto Step 3b and repeat the search procedure.

   Hereafter, we call the pair of nodes $(\bar{x}_i, \bar{y}_i)$ in $((\bar{x}_i, \bar{y}_i), \bar{m}_i) \in \mathcal{S}$ a pair of target nodes.
4. Run Alg.5 with a parameter t to develop 2^t nodes in $\mathcal{FG}_{f_1}$ (resp. $\mathcal{FG}_{f_2}$), and store them in a data structure $\mathcal{G}_1$ (resp. $\mathcal{G}_2$). Moreover,
   (a) Store at each node its distance from a particular target node (say target node $\bar{x}_1$ (resp. $\bar{y}_1$), together with its distance from the cycle (*i.e.*, its height, similar to Phase 3 in Section 5 of [PW14]).
   (b) Store the distance of other target nodes $\bar{x}_i$ (resp. $\bar{y}_i$) to this particular target node $\bar{x}_1$ (resp. $\bar{y}_1$) in a table $\mathcal{T}_x$ (resp. $\mathcal{T}_y$) by iterating f_1 (resp. f_2) along the cycle.
   (c) Thus, when the distance of a node from the particular target node and that from the cycle is known from $\mathcal{G}_1$ (resp. $\mathcal{G}_2$), the distances of this node from all the other target nodes can be immediately deduced from $\mathcal{T}_x$ (resp. $\mathcal{T}_y$). Specifically, suppose the distance of a node x_0 from $\bar{x}_1$ is d_1 and its height is e_1, and suppose the distance of a target node $\bar{x}_i$ from $\bar{x}_1$ is d_i; then, the distance of x_0 from $\bar{x}_i$ is $d_1 - d_i$ if $d_i \leq (d_1 - e_1)$, and $L_1 - d_i + d_1$ if $d_i > (d_1 - e_1)$.
5. Find a message $M_{\texttt{Link}}$ that maps $(\hat{x}, \hat{y})$ to a pair of target nodes $(\bar{x}, \bar{y})$ for some $((\bar{x}, \bar{y}), \bar{m}) \in \mathcal{S}$. We search for such a linking message among a set of special messages: $M_{\texttt{Link}} = \hat{m} \,\|\, m \,\|\, m \,\|\, \cdots \,\|\, m$, where $\hat{m}$ is a random single-block message, and m is the fixed message at Step 2. The search procedure is as follows.

(a) Select a random message block w, and compute $x_0 = h_1(\hat{x}, w)$ and $y_0 = h_2(\hat{y}, w)$.
(b) Compute a chain by iteratively applying f_1 (resp. f_2) to update x_0 (resp. y_0) until either of the following two cases occurs.
- The chain length reaches 2^{n-t}. In this case, goto Step 5a;
- The chain encounters a node stored in $\mathcal{G}_1$ (resp. $\mathcal{G}_2$). In this case, compute the distance of x_0 (resp. y_0) to every target node $\bar{x}_i$ (resp. $\bar{y}_i$) as described in Step 4c, and denote it as $d_{\bar{x}_i}$ (resp. $d_{\bar{y}_i}$). Examine whether $(d_{\bar{x}_i} - d_{\bar{y}_i} \mod L_1)$ is in $\mathcal{D}$. If it is, set $d_1 \triangleq d_{\bar{x}_i}$ and $d_2 \triangleq d_{\bar{y}_i}$, derive the corresponding j and k such that $d_1 + j \cdot L_1 = d_2 + k \cdot L_2$ holds. Set $q \triangleq d_1 + j \cdot L_1 = d_2 + k \cdot L_2$, and then $M_{\texttt{Link}} \triangleq \hat{m} \parallel [m]^q$. Besides, set $\hat{m} \triangleq w$ and $\bar{m} \triangleq \bar{m}_i$. Otherwise, goto Step 5a.

6. Derive a message $M_{\parallel L-q-3}$ with a block length of $L - q - 3$ from the expandable message $\mathcal{M}_{\texttt{SEM}}$.
7. Produce a preimage $\boldsymbol{M}$ of the target hash digest V as follows:

$$\begin{aligned} \boldsymbol{M} &\triangleq M_{\parallel L-q-3} \parallel M_{\texttt{Link}} \parallel \bar{m} \parallel pad \\ &= M_{\parallel L-q-3} \parallel \hat{m} \parallel [m]^q \parallel \bar{m} \parallel pad. \end{aligned}$$

Complexity Analysis. For parameters $L \geq 2^{n/2}$, $s \geq 0$ and $t \geq 0$, the complexity of the steps of the attack is given below (ignore the constant and polynomial factors for simplicity of description).

- **Step 1:** $L + n^2 \cdot 2^{n/2}$ (refer to Sect. 2.6);
- **Step 2:** $2^{n/2} + L/L_1 \approx 2^{n/2} + 2^{-n/2} \cdot L \approx 2^{n/2}$;
- **Step 3:** $2^{s+n/2}$;
  One execution of the search procedure has a complexity of $L_1 + L_2$, and contributes to $L_1 \cdot L_2$ pairs. As $L_1 \cdot L_2 = \Theta(2^n)$, one tuple can be obtained by a constant number of executions. Hence, the number of necessary executions is $\Theta(2^s)$, and the complexity of this step is $\Theta(2^{s+n/2})$.
- **Step 4:** $2^t + 2^{n/2}$;
  The complexity of developing 2^t nodes and computing their distance to a particular target node is 2^t (refer to Alg. 5 and Step 4a). The complexity to compute the distance of all the other target nodes to the particular target node is upper bounded by $2^{n/2}$ (refer to the expectation of the maximum cycle length in Thm. 3). Hence, the complexity of this step is $2^t + 2^{n/2}$.
- **Step 5:** $2^{2n-t-s}/L$;
  One execution of the search procedure requires a time complexity of 2^{n-t}. Clearly, a constant factor of both chains encounter nodes stored in $\mathcal{G}_1$ and $\mathcal{G}_2$. We mainly need to evaluate the probability of deriving a common distance for each chain. For every pair of target nodes $(\bar{x}_i, \bar{y}_i)$, the value of $d_{x_i} - d_{y_i}$ is equal to a correctable distance bias in $\mathcal{D}$ with a probability of $\#C \cdot 2^{-n/2} \approx L \cdot 2^{-n}$. Since there are 2^s pairs of target nodes, the success probability of each chain

is $L \cdot 2^{s-n}$. Hence, the total number of chains is $2^{n-s}/L$, and the complexity of this step is $2^{n-t} \cdot 2^{n-s}/L = 2^{2n-t-s}/L$.
- **Steps 6 and 7:** $\mathcal{O}(L)$.

The overall complexity is computed as (denote $L = 2^\ell$)

$$2^\ell + 2^{s+n/2} + 2^t + 2^t + 2^{n/2} + 2^{2n-t-s-\ell},$$

where the complexity of Step 2 is ignored.

Now, we search for parameters t and s that give the lowest complexity. First, we balance Step 3 and Step 4 by setting $s + n/2 = t$. That gives $s = t - n/2$. The complexity becomes (ignoring constant factors) $2^\ell + 2^t + 2^{5n/2-2t-\ell}$. We then make a balance by setting $t = 2n - 2t + n/2 - \ell$, *i.e.*, $t = 5n/6 - \ell/3$. Thus, the total complexity becomes

$$2^\ell + 2^{5n/6-\ell/3}.$$

Hence,

- for the case $n/2 \le \ell \le 5n/8$, the final complexity is $2^{5n/6-\ell/3}$;
- for the case $5n/8 < \ell$, the final complexity is 2^ℓ.

The optimal complexity is $2^{5n/8}$, obtained for messages of length $2^{5n/8}$ (see Fig. 12 for a trade-off curve).

5.3 Optimizing the Multi-Cycles-Based Preimage Attack on the XOR Combiner Using the Interchange Structure

Again, similar to the previous deep-iterates-based preimage attack, this multi-cycles-based preimage attack on the XOR combiner can also be improved using an interchange structure. The complexity of the attack using a 2^r-interchange structure is analysed as follows.

Denote $L = 2^\ell$. For parameters $t \ge n/2$, $\ell \ge n/2$ and $0 \le r \le \ell/2$ (because the length 2^{2r} of the interchange structure should be no larger than the message length $L = 2^\ell$), the complexity of each step is as follows:

Step 1: $2^\ell + n^2 \cdot 2^{n/2}$ **Step 2:** $2^{n/2} + 2^{\ell-n/2}$ **Step 3:** $2^{s+n/2}$

Step 4: $2^t + 2^{n/2}$ **Step 5:** $2^{n-t+r} \cdot 2^{n-s-2r-\ell} + 2^{n/2+2r}$ **Step 6:** 2^ℓ

The sum of dominant terms is

$$2^\ell + 2^{s+n/2} + 2^t + 2^{2n-t-s-r-\ell} + 2^{n/2+2r}.$$

We first balance the last four terms by setting $s + n/2 = t = 2n - t - s - r - \ell = n/2 + 2r$. Thus, $t = 11n/14 - 2\ell/7$, $s = 2n/7 - 2\ell/7$, and $r = n/7 - \ell/7$. Note that for $\ell \ge n/2$, we have $n/7 - \ell/7 \le n/14 < n/2$. Thus, the restriction $r \le \ell/2$ always holds in this setting. The total complexity turns to be

$$2^\ell + 2^{11n/14-2\ell/7}.$$

Hence,

– for the case $n/2 \leq \ell \leq 11n/18$, the final complexity is $2^{11n/14-2l/7}$;
– for the case $11n/18 < \ell$, the final complexity is 2^{ℓ}.

The optimal complexity is $2^{11n/18}$, obtained for messages of length $2^{11n/18}$ (see Fig. 12 for a trade-off curve).

6 Second-Preimage Attack on Concatenation Combiners Based on Deep Iterates

In this section, we introduce the first second-preimage attack faster than 2^n on concatenation combiners of MD hashes.

In this attack, we are given a challenge message $\boldsymbol{M} = m_1 \parallel m_2 \parallel \ldots \parallel m_L$, and our goal is to find another message $\boldsymbol{M'}$ such that $\mathcal{H}_1(\boldsymbol{M'}) \parallel \mathcal{H}_2(\boldsymbol{M'}) = \mathcal{H}_1(\boldsymbol{M}) \parallel \mathcal{H}_2(\boldsymbol{M})$ (or equivalently $\mathcal{H}_1(\boldsymbol{M'}) = \mathcal{H}_1(\boldsymbol{M})$ and $\mathcal{H}_2(\boldsymbol{M'}) = \mathcal{H}_2(\boldsymbol{M})$). We denote the sequence of internal states computed during the invocation of h_1 (respectively, h_2) on $\boldsymbol{M}$ by $a_0, a_1, \ldots, a_L$ (respectively, $b_0, b_1, \ldots, b_L$).

The general framework of our attack is similar to the one of the long message second-preimage attack on a single MD hash proposed by Kelsey and Schneier and described in Sect.2.2. Namely, we first compute the sequences of internal states $a_1, \ldots, a_L$ and $b_1, \ldots, b_L$ by applying the compression functions h_1 and h_2 on the challenge message $\boldsymbol{M} = m_1 \parallel \ldots \parallel m_L$. Our goal is then to "connect" to one of the state pairs (a_i, b_i) using a different message prefix of the same size. Once we manage to achieve this, we can reuse the same message suffix as in $\boldsymbol{M}$ and obtain a second preimage.

There are two main challenges in this approach, where the main challenge is to connect to some state pair (a_i, b_i) generated by $\boldsymbol{M}$ from a different message. The secondary challenge is to ensure that the connected message prefixes are of the same length. We overcome the secondary challenge by building a simultaneous expandable message for two Merkle-Damgård hash functions, as described in Sect. 2.6.

A much more difficult challenge is to actually connect to the challenge message on a state pair (a_i, b_i) from a different message of arbitrary (smaller) length. Indeed, the obvious approach of attempting to reach an arbitrary $2n$-bit state pair by trying random messages requires more than 2^n time, since the number of target state pairs is equal to the message length which is smaller than 2^n. A more promising approach is to use the *interchange structure* introduced in Sect.3. Recall that, the interchange structure consists of an initial state pair (a, b), a set of message fragments $\mathcal{M}$ and two sets of internal states $\mathcal{A}$ (for $\mathcal{H}_1$) and $\mathcal{B}$ (for $\mathcal{H}_2$) such that for any value $A \in \mathcal{A}$ and any value $B \in \mathcal{B}$, it is possible to efficiently construct a message $M_{A,B} \in \mathcal{M}$ that maps (a, b) to (A, B). Assume that there exists an index $i \in \{1, 2, \ldots, L\}$ such that $a_i \in \mathcal{A}$ and $b_i \in \mathcal{B}$; then, we can connect to (a_i, b_i) using M_{a_i,b_i} as required. Unfortunately, this does not result in an efficient attack, essentially because the complexity of building an interchange structure for sufficiently large sets $\mathcal{A}$ and $\mathcal{B}$ is not efficient enough.

Here, as in the deep-iterate-based preimage attack on XOR combiner, we use deep iterates in functional graphs of $f_1(\cdot) \triangleq h_1(\cdot, m)$ and $f_2 \triangleq h_2(\cdot, m)$ (as a

result, it is not applicable when any one of the underlying hash functions are of the HAIFA framework). More specifically, our goal is to find a state pair (a_p, b_p) composed of two deep iterates in $\mathcal{FG}_{f_1}$ and $\mathcal{FG}_{f_2}$, respectively.[21] Once we find such a "special" state pair, we show how to simultaneously reach both of its states in an efficient manner from an arbitrary state pair. Combined with the simultaneous expandable message, this gives the desired second preimage.

Next, as in previous attack, we start by providing a high-level overview of the attack and then give technical details.

6.1 Overview of the Attack

The attack is composed of three main phases.

Attack 4: Second-preimage attack on the concatenation combiner based on deep iterates

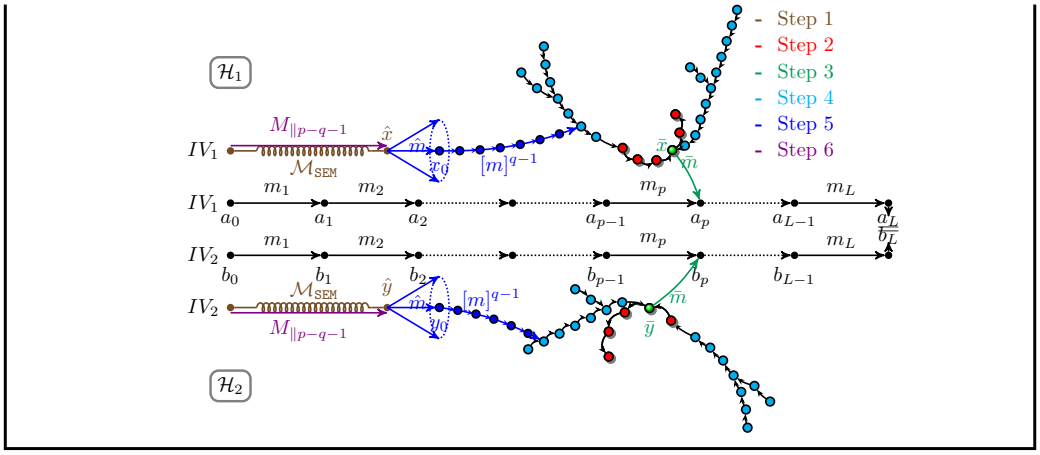
- **Phase 1:** Build a simultaneous expandable message $\mathcal{M}_{\texttt{SEM}}$ for $\mathcal{H}_1$ and $\mathcal{H}_2$ as described in Sect.2.6, starting from the initial state pair (IV_1, IV_2) and ending at a final state pair $(\hat{x}, \hat{y})$.
- **Phase 2:** Find a pair of states $(\bar{x}, \bar{y})$, a message block $\bar{m}$ and an index p such that $(\bar{x}, \bar{y}) \xrightarrow{\bar{m}} (a_p, b_p)$ (note that the state pair (a_p, b_p) is computed during the evaluation of the challenge message). Moreover, the state pair $(\bar{x}, \bar{y})$ should have the special property that will be defined in the detailed description of this phase.
- **Phase 3:** Start from $(\hat{x}, \hat{y})$ and compute a message fragment $\hat{M}_{\|q}$ (shorter than $p-1$) such that $(\hat{x}, \hat{y}) \xrightarrow{\hat{M}_{\|q}} (\bar{x}, \bar{y})$. This phase can be performed efficiently due to the special property of $(\bar{x}, \bar{y})$.

To compute the second preimage, we pick $M_{\|p-q-1}$ from the simultaneous expandable message $\mathcal{M}_{\texttt{SEM}}$, giving $(IV_0, IV_1) \xrightarrow{M_{\|p-q-1}} (\hat{x}, \hat{y})$, and concatenate $M_{\|p-q-1} \,\|\, \hat{M}_{\|q} \,\|\, \bar{m}$ in order to reach the state pair (a_p, b_p) from (IV_1, IV_2) with a message of appropriate length p. Indeed, we have

$$(IV_0, IV_1) \xrightarrow{M_{\|p-q-1}} (\hat{x}, \hat{y}) \xrightarrow{\hat{M}_{\|q}} (\bar{x}, \bar{y}) \xrightarrow{\bar{m}} (a_p, b_p).$$

Altogether, we obtain the second preimage: $\boldsymbol{M'} = M_{\|p-q-1} \,\|\, \hat{M}_{\|q} \,\|\, \bar{m} \,\|\, m_{p+1} \,\|\, \ldots \,\|\, m_L$.

[21] The actual attack is slightly different, as it searches for deep iterates from which (a_p, b_p) can be reached with a common message block.



Complexity Analysis. Denote $L = 2^\ell$. For a parameter $g_1 \geq max(n/2, n-\ell)$, the complexity of the phases of the attack (as computed in their detail description) is given below (ignoring constant factors).

Phase 1: $L + n^2 \cdot 2^{n/2} = 2^\ell + n^2 \cdot 2^{n/2}$ **Phase 2:** $1/L \cdot 2^{2n-g_1} = 2^{2n-g_1-\ell}$

Phase 3: $2^{3g_1/2}$

We balance the second and third phases by setting $2n - g_1 - \ell = 3g_1/2$, or $g_1 = 2/5 \cdot (2n - \ell)$, giving time complexity of $2^{3/5 \cdot (2n-\ell)}$. This trade-off holds as long as $2^\ell + n^2 \cdot 2^{n/2} \leq 2^{3/5(2n-\ell)}$, or $\ell \leq 3n/4$. The optimal complexity is $2^{3\ell/4}$, obtained for $\ell = 3n/4$ (see Fig. 12 for a trade-off curve). The attack is faster than 2^n (Joux's preimage attack) for[22] $\ell > n/3$. The message range for which the attack is faster than 2^n can be slightly improved to $L \geq 2^{2n/7}$ using the optimized attack, described in Sect. 6.3.

6.2 Details of the Second-Preimage Attack on Concatenation Combiners using Deep Iterates

Details of Phase 1 can be found in Sect. 2.6.

Details of Phase 2: Finding a Target State Pair. In the second phase, we fix an arbitrary message block m, giving rise to the functional graph $\mathcal{FG}_{f_1}$ of $f_1(\cdot) \triangleq h_1(\cdot, m)$ and $\mathcal{FG}_{f_2}$ of $f_2(\cdot) \triangleq h_2(\cdot, m)$ and let $g_1 \geq n/2$ be a parameter (to be determined later). Our goal is to find a pair of states $(\bar{x}, \bar{y})$, a message block $\bar{m}$ and an index p such that the following two conditions hold:

1. The state $\bar{x}$ is a 2^{n-g_1}-th iterate in $\mathcal{FG}_{f_1}$ and $\bar{y}$ is a 2^{n-g_1}-th iterate in $\mathcal{FG}_{f_2}$.
2. The state pair $(\bar{x}, \bar{y})$ is mapped to (a_p, b_p) by $\bar{m}$, or $(\bar{x}, \bar{y}) \xrightarrow{\bar{m}} (a_p, b_p)$.

The algorithm of this phase is given below.

[22] Note that for $\ell > n/3$, $g_1 = 2/5 \cdot (2n - \ell) > 2n/3 > max(n/2, n - \ell)$, as required.

Phase 2 of Attack 4: Finding a target state pair

1. Fix an arbitrary single-block value m and get $f_1(\cdot) \triangleq h_1(\cdot, m)$ and $f_2(\cdot) \triangleq h_2(\cdot, m)$.
2. Expand $\mathcal{FG}_{f_1}$ using Alg. 5 with parameter g_1. Store all encountered 2^{n-g_1}-th iterates in a table $\mathcal{T}_1$.
3. Expand $\mathcal{FG}_{f_2}$ using Alg. 5 with parameter g_1. Store all encountered 2^{n-g_1}-th iterates in a table $\mathcal{T}_2$.
4. For single-block values $m' = 0, 1, \ldots$, perform the following steps:
   (a) For each node $x \in \mathcal{T}_1$, evaluate $x' = h_1(x, m')$ and store the matches $x' = a_i$ with the[a] sequence $a_1, \ldots, a_L$ in a table $\mathcal{T}_1'$, sorted according to the index i of a_i.
   (b) Similarly, for each node $y \in \mathcal{T}_2$ evaluate $y' = h_2(y, m')$ and look for matches $y' = b_j$ with the sequence $b_1, \ldots, b_L$. For each match with some b_j, search for the index j in the table $\mathcal{T}_1'$. If a match $i = j$ is found, set $p \triangleq i$ (namely, $(a_p, b_p) \triangleq (x', y')$), $\bar{m} \triangleq m'$ and $(\bar{x}, \bar{y}) \triangleq (x, y)$. This gives $(\bar{x}, \bar{y}) \xrightarrow{\bar{m}} (a_p, b_p)$, as required. Otherwise, (no match $i = j$ is found), go back to Step 4.

[a] More precisely, due to the minimal-length restriction of the expandable message, matches $x' = a_i$ with i less than (approximately) $C^2 \approx n^2$ cannot be exploited in the full attack. Moreover, the maximal exploitable value of i is $L - 2$. However, the fraction of these nodes is very small and can be ignored in the complexity analysis.

The time complexity of steps 2 and 3 (which execute the Alg. 5) is approximately 2^{g_1}. The time complexity of step 4.(a) and step 4.(b) is also bounded by 2^{g_1} (given that $a_1, \ldots, a_L$ and $b_1, \ldots, b_L$ are sorted in memory), as the size of $\mathcal{T}_1$ and $\mathcal{T}_2$ is at most 2^{g_1} and the number of matches found in each step can only be smaller.

We now calculate the expected number of executions of Step 4 until the required (a_p, b_p) is found. According to Observation 1 in Sect. 2.7, the expected size of $\mathcal{T}_1$ and $\mathcal{T}_2$ (that contain iterates of depth 2^{n-g_1}) is close to 2^{g_1}. According to the birthday paradox, the expected size of $\mathcal{T}_1'$ is approximately $L \cdot 2^{g_1-n}$. Similarly, the number of matches $y' = b_j$ is also approximately $L \cdot 2^{g_1-n}$. The probability of a match $i = j$ in Step 4.(b) is computed using a birthday paradox on the L possible indexes, namely, $1/L \cdot (L \cdot 2^{g_1-n})^2 = L \cdot 2^{2g_1-2n}$. As a result, Step 4 is executed approximately $1/L \cdot 2^{2n-2g_1}$ times until the required (a_p, b_p) is found (the executions with different blocks m' are essentially independent). Altogether, the total time complexity of this step is

$$2^{g_1} \cdot 1/L \cdot 2^{2n-2g_1} = 1/L \cdot 2^{2n-g_1}.$$

Since the index p is uniformly distributed in the interval $[1, L]$, we will assume that $p = \Theta(L)$.

Details of Phase 3: Hitting the Target State Pair. In the third and final phase, we start from the pair of endpoints $(\hat{x}, \hat{y})$ of the simultaneous expandable message constructed in Phase 1 and compute a message fragment $\hat{M}_{\|q}$ of length $q < p - 1$ such that $(\hat{x}, \hat{y}) \xrightarrow{\hat{M}_{\|q}} (\bar{x}, \bar{y})$. Like in the deep-iterates-based preimage attack on the XOR combiner, here, we again use in a strong way the fact that the state $\bar{x}$ (and $\bar{y}$) is a deep iterate (of depth 2^{n-g_1}) in the functional graph of $f_1(x)$ ($f_2(y)$).

This phase is carried out by picking an arbitrary starting message block $\hat{m}$, which gives points $x_0 = h_1(\hat{x}, \hat{m})$ and $y_0 = h_2(\hat{y}, \hat{m})$. We then continue to evaluate the chains $x_i = h_1(x_{i-1}, m)$ and $y_j = h_2(y_{j-1}, m)$ up to a maximal length $L' = 2^{n-g_1}$. We hope to encounter $\bar{x}$ at some distance $q-1$ from x_0 and to encounter $\bar{y}$ at the same distance $q-1$ from y_0. Given that $q-1 < p$, this will give the required $\hat{M} = \hat{m} \,\|\, [m]^{q-1}$ (where $[m]^{q-1}$ denotes the concatenation of $q-1$ message blocks m), which is of length $q < p-1$. If $\bar{x}$ and $\bar{y}$ are encountered at different distances in the chains, or at least one of them is not encountered at all, we pick a different value for $\hat{m}$ and start again.

Since $\bar{x}$ (resp. $\bar{y}$) is an iterate of depth 2^{n-g_1} in $\mathcal{FG}_{f_1}$ (resp. $\mathcal{FG}_{f_2}$), it is an endpoint of a chain of states of length $L' = 2^{n-g_1}$ (such a chain was computed in Phase 2). According to Observation 2, the probability that $\bar{x}$ and $\bar{y}$ will be encountered at the same distance from arbitrary starting points x_0 and y_0 of chains is 2^{n-3g_1}. The probability calculation gives rise to the conclusion that we need to compute approximately 2^{3g_1-n} chains from different starting points. Each chain is of length up to $L' = 2^{n-g_1}$. This gives a total time complexity of about $2^{3g_1-n+n-g_1} = 2^{2g_1}$. Since $g_1 \geq n/2$, the time complexity of the full algorithm is at least 2^n, and the attack is not faster than Joux's preimage attack.

To optimize the algorithm, as we did in the deep-iterates-based preimage attack, we use a look-ahead procedure by further expanding the graphs of f_1 and f_2. The difference is that since we only select a single pair of deep iterate as the target, we do not use the distinguished point technique[23] [24]. We pick a parameter $g_2 > g_1$ and execute the algorithm below.

Phase 3 of Attack 4: Hitting the target state pair

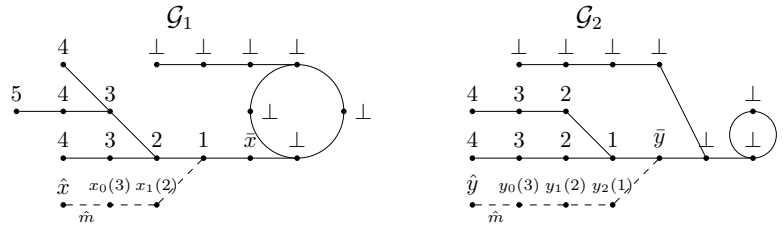
1. Develop 2^{g_2} nodes in $\mathcal{FG}_{f_1}$ (and $\mathcal{FG}_{f_2}$) (as specified in Alg. 5) with the following modifications.

[23] One may ask why we did not compute a larger set $\mathcal{S}$, as we did in Phase 3 of Attack 2. The reason for this is that it can be shown that in this case, a set of size 1 is optimal.

[24] One may also ask why we did not use cyclic nodes and multi-cycles to further improve this second-preimage attack on concatenation combiners as we did for the preimage attack on XOR combiners. The reason is that optimization on Phase 2 of Attack 4 has reached it limitation because of the limited number of candidate state pairs for $(\bar{x}, \bar{y})$. Thus, the complexity of Phase 2 becomes the bottleneck and cannot be improved using cyclic nodes.

- Store at each node its distance from $\bar{x}$ in $\mathcal{FG}_{f_1}$ (or $\bar{y}$ in $\mathcal{FG}_{f_2}$) (the maximal stored distance is $L' = 2^{n-g_1}$): for each computed chain, once it hits a previously visited node in the graph, obtain its stored distance from $\bar{x}$ (or $\bar{y}$) and update it in all the computed nodes in the current chain up to the maximal value $L' = 2^{n-g_1}$.
- If a chain does not hit $\bar{x}$, then the distance of its nodes is undefined and stored as a special value $\perp$. This special value is also used for nodes whose distance from $\bar{x}$ is greater than L'.
- The evaluated nodes for $\mathcal{FG}_{f_1}$ ($\mathcal{FG}_{f_2}$) are stored in a data structure $\mathcal{G}_1$ ($\mathcal{G}_2$).

2. For single-block values $\hat{m} = 0, 1, \ldots$, compute $x_0 = h_1(\hat{x}, \hat{m})$ and $y_0 = h_2(\hat{y}, \hat{m})$ and repeat the following step:
   (a) Compute the chains $\vec{x}$ and $\vec{y}$ up to maximal length $L' = 2^{n-g_1}$ or until they hit $\mathcal{G}_1$ and $\mathcal{G}_2$ (respectively).
   - If $\vec{x}$ (or $\vec{y}$) does not hit $\mathcal{G}_1$ ($\mathcal{G}_2$), return to Step 2.
   - Otherwise, once $\vec{x}$ ($\vec{y}$) hits $\mathcal{G}_1$ ($\mathcal{G}_2$), obtain the stored distance from $\bar{x}$ ($\bar{y}$) at the collision point. If the distance to $\bar{x}$ (or $\bar{y}$) is undefined, return to Step 2.
   - Compute the respective distances i and j of x_0 and y_0 from $\bar{x}$ and $\bar{y}$. If $i \neq j$, return to Step 2.
   - Otherwise ($i = j$), denote $q = i + 1$. If $q \geq p - 1$, return to Step 2.
   - Otherwise ($q < p - 1$), return the message $\hat{M} = \hat{m} \parallel [m]^i = \hat{m} \parallel [m]^{q-1}$ as output.



The time complexity of Step 1 is approximately 2^{g_2}. As concluded above, in Step 2, we perform approximately 2^{3g_1-n} trials on the value of $\hat{m}$ before finding two starting points x_0 and y_0 at the same distance from $\bar{x}$ and $\bar{y}$. According to the analysis of Sect. 2.7, each trial requires approximately 2^{n-g_2} computation (before hitting $\mathcal{G}_1$ and $\mathcal{G}_2$). Therefore, the total time complexity of this phase is $2^{g_2} + 2^{3g_1-n} \cdot 2^{n-g_2} = 2^{g_2} + 2^{3g_1-g_2}$. The complexity is minimized by setting $g_2 = 3g_1/2$ which balances the two terms and gives a time complexity of

$$2^{3g_1/2}.$$

Finally, we note that the memory complexity of this algorithm can be optimized using distinguished points. A detailed way to achieve this has been presented in the closely related algorithm in Sect. 4.2.

6.3 Optimizing the Deep-Iterates-Based Second-Preimage Attack on Concatenation Combiners using an Interchange Structure

Similar to the deep-iterates-based preimage attack on the XOR combiner, this deep-iterates-based second-preimage attack can also be slightly improved using an interchange structure. The detailed complexity analysis of the attack using a 2^r-interchange structure is as follows:

Denote $L = 2^\ell$. For parameters $g_1 \geq \max(n/2, n-\ell)$, $g_2 \geq 0$, and $0 \leq r \leq \ell/2$ (because the length 2^{2r} of the interchange structure should be less than the message length 2^ℓ), the complexity of the phases of the attack is (ignoring constant factors)

Phase 1: $2^\ell + n^2 \cdot 2^{n/2}$ **Phase 2:** $2^{2n-g_1-\ell}$

Phase 3: $2^{g_2} + 2^{3g_1-n-2r} \cdot 2^{r+n-g_2} + 2^{n/2+2r} = 2^{g_2} + 2^{3g_1-g_2-r} + 2^{n/2+2r}$

Compared with the complexity of the attack in Sect.6.1, the difference lies in Phase 3. In the complexity formula of Phase 3, the term 2^{g_2} is the number of nodes developed in the look-ahead procedure; the term 2^{3g_1-n-2r} is the required number of samplings on the random massage block $\hat{m}$ when trying to find a pair of starting nodes reaching the pair of 2^{n-g_1}-th iterates $(\bar{x}, \bar{y})$ at a common distance; the term 2^{r+n-g_2} is the time complexity for computing the distances of a set of 2^r starting nodes (generated using the same value for $\hat{m}$) from a target node; and the term $2^{n/2+2r}$ is the time complexity for building a 2^r-interchange structure.

We first balance the first two terms in Phase 3 by setting $g_2 = 3g_1 - g_2 - r$, which gives $g_2 = 3g_1/2 - r/2$. The sum of all dominant terms is

$$2^\ell + 2^{2n-g_1-\ell} + 2^{3g_1/2-r/2} + 2^{n/2+2r}.$$

For $\ell > 7n/17$, we set $2n - g_1 - \ell = 3g_1/2 - r/2 = n/2 + 2r$, which gives $g_1 = 19n/22 - 5\ell/11$ and $r = 7n/22 - 3\ell/11$. The total complexity is then $2^\ell + 2^{25n/22-6\ell/11}$. For $\ell > 25n/34$, we have $25n/22 - 6\ell/11 < \ell$. Thus, the time complexity is 2^ℓ for $\ell > 25n/34$. Note that, $g_1 = 19n/22 - 5\ell/11$ fulfils the restriction $g_1 \geq \max(n/2, n-\ell)$ for $n/4 \leq \ell \leq 4n/5$, and $r = 7n/22 - 3\ell/11$ fulfils the restriction $2r < l$ as long as $\ell > 7n/17$. Thus, the time complexity is $2^{25n/22-6\ell/11}$ for $7n/17 < \ell \leq 25n/34$.

For $\ell \leq 7n/17$, we directly set $r = \ell/2$ (the maximum under the restriction $2r < \ell$) to optimize the complexity (because as shown next, the balanced sum $2^{2n-g_1-\ell} + 2^{3g_1/2-r/2}$ is greater than $2^{n/2+2r}$ under the restrictions $g_1 \geq \max(n/2, n-\ell)$, $2r < \ell$ and $\ell \leq 7n/17$). The formula is $2^{2n-g_1-\ell} + 2^{3g_1/2-\ell/4} + 2^{n/2+\ell}$. We balance the first two terms by setting $2n - g_1 - \ell = 3g_1/2 - \ell/4$, from which we deduced that $g_1 = 4n/5 - 3\ell/10$ (fulfills the restriction $g_1 \geq \max(n/2, n-\ell)$ as long as $\ell > 2n/7$). Then, the complexity is $2^{6n/5-7\ell/10} + 2^{n/2+\ell}$. Since $6n/5 - 7\ell/10 \geq n/2 + \ell$ for $\ell \leq 7n/17$, the total time complexity is then $2^{6n/5-7\ell/10}$. It is no less than 2^n for $\ell < 2n/7$.

Thus, the final time complexity of this attack using interchange structure is summarized as follows:

- For the case $\ell < 2n/7$, the complexity is 2^n achieved by Joux's attack;
- For the case $2n/7 \leq \ell \leq 7n/17$, the complexity of this attack is $2^{6n/5-7\ell/10}$;
- For the case $7n/17 < \ell \leq 25n/34$, the complexity of this attack is $2^{25n/22-6\ell/11}$;
- For the case $\ell > 25n/34$, the complexity of this attack is 2^{ℓ}.

The optimal complexity is $2^{25n/34}$, obtained for messages of length $2^{25n/34}$ (see Fig. 12 for a trade-off curve).

7 Second-Preimage Attack on the Zipper Hash

In this section, we present the first second-preimage attack on the Zipper hash, which is applicable for *idealized* compression functions and hence a generic attack. Again, the attack is based on the deep iterates and multi-cycles in the functional graphs defined by $f_1(\cdot) \triangleq h_1(\cdot, m)$ and $f_2(\cdot) \triangleq h_2(\cdot, m)$ with a fixed single-block message value m. The general framework is similar to that of the above ones on combiners of MD hashes. However, some special specifications on the Zipper hash allow the attacker to choose an optimal configuration for the attack and to launch a more efficient connecting phase in the attack. More precisely, as opposed to the two parallel combiners, in the Zipper hash, the message length is placed in the middle of the two passes. Thus, when we first connect our crafted message to the challenge message on an internal state in the second pass, the message prefix of our crafted message is fixed. This prefix does not include the length padding. As a result, the length of our crafted message is not necessarily equal to the length of the challenge message. Thus, we can choose a proper length for our crafted message that optimizes the attack complexity. A further uniqueness of the Zipper hash is that its second pass processes the message blocks in a reversed order. Thus, in the attack, when looking for a pair of nodes $(\check{x}, \check{y})$ reaching two predefined nodes of deep iterates $(\bar{x}, \bar{y})$ at a common distance, $\check{x}$ and $\check{y}$ are computed with different message blocks. This enables us to launch an efficient meet-in-the-middle procedure during the connecting phase. Accordingly, Joux's multi-collision (see Sect.2.1) is used to facilitate the meet-in-the-middle procedure, and the previous simultaneous expandable message in Sect.2.6 is fine-tuned to adapt to the Zipper hash.

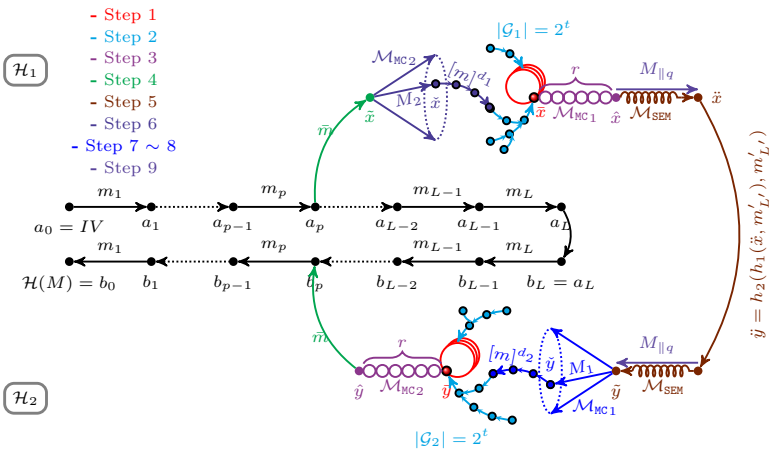
7.1 Overview of the Attack

Given a message $\boldsymbol{M} = m_1 \| \cdots \| m_L$, the goal of the second-preimage attack on the Zipper hash is to find another message $\boldsymbol{M'}$ such that $\mathcal{H}_2(\mathcal{H}_1(IV, \boldsymbol{M}), \overleftarrow{\boldsymbol{M}}) = \mathcal{H}_2(\mathcal{H}_1(IV, \boldsymbol{M'}), \overleftarrow{\boldsymbol{M'}})$, where $\overleftarrow{\boldsymbol{M}}$ is a message generated by reversing the order of message blocks of $\boldsymbol{M}$ (we call $\overleftarrow{\boldsymbol{M}}$ the reverse of $\boldsymbol{M}$), *i.e.*, $\overleftarrow{\boldsymbol{M}} = m_L \| m_{L-1} \| \cdots \| m_1$, and $\overleftarrow{\boldsymbol{M'}}$ is the reverse of $\boldsymbol{M'}$. Here, we briefly list the *main* steps of the attack.

Attack 5: Second-preimage attack on the Zipper hash

- **Phase 1:** Get a cyclic node $\bar{x}$ (resp. $\bar{y}$) located in the largest component of $\mathcal{FG}_{f_1}$ (resp. $\mathcal{FG}_{f_2}$); Get the cycle length L_1 (resp. L_2), and compute the set of correctable distance bias $\mathcal{D} = \{i \cdot \Delta L \mod L_1 \mid i = 0, 1, \ldots, \#C\}$, where $\Delta L = L_2 \mod L_1$ and $\#C$ is the number of cycles to the maximum. After that, generate a Joux's multi-collision $\mathcal{M}_{\texttt{MC1}}$ (resp. $\mathcal{M}_{\texttt{MC2}}$) starting from $\bar{x}$ (resp. $\bar{y}$) and denote its final endpoints by $\hat{x}$ (resp. $\hat{y}$).
- **Phase 2:** Build a cascade simultaneous expandable message $\mathcal{M}_{\texttt{SEM}}$ across the two passes starting from $\hat{x}$ and denote its final endpoint by $\tilde{y}$.
- **Phase 3:** Find a message block $\bar{m}$ mapping the final endpoint $\hat{y}$ of the second Joux's multi-collision $\mathcal{M}_{\texttt{MC2}}$ to one of the chaining states b_p in the second pass of the original message; Then, in the first pass, use $\bar{m}$ to update the corresponding chaining state a_p to a state $\tilde{x}$.
- **Phase 4:** Find a message $M_{\texttt{Link}}$ such that $(\tilde{x}, \tilde{y}) \xrightarrow{M_{\texttt{Link}}} (\bar{x}, \bar{y})$[a].
  This is a meet-in-the-middle procedure. First, exploit the messages in the two Joux's multi-collisions $\mathcal{M}_{\texttt{MC1}}$ and $\mathcal{M}_{\texttt{MC2}}$ to map $(\tilde{x}, \tilde{y})$ to two independent sets of starting nodes, compute and store their distances from target node $\bar{x}$ and $\bar{y}$ in two tables $\mathcal{T}_1$ and $\mathcal{T}_2$ independently. Then, find a match between $\mathcal{T}_1$ and $\mathcal{T}_2$ (check whether the difference between some stored distances is correctable by values in $\mathcal{D}$), denote the matched distances by d_1 and d_2, set the common distance $d \triangleq d_1 + i \cdot L_1 = d_2 + j \cdot L_2$ for some i and j, and retrieve the corresponding messages $M_2 \in \mathcal{M}_{\texttt{MC2}}$ and $M_1 \in \mathcal{M}_{\texttt{MC1}}$ that generate the corresponding starting nodes.

At the end, select a message suffix $M_{\|q}$ with a proper block length $q = L' - p - 2r - d$ from the simultaneous expandable message $\mathcal{M}_{\texttt{SEM}}$ and construct a second preimage $\boldsymbol{M'}$: $m_1 \,\|\, \cdots \,\|\, m_p \,\|\, M_2 \,\|\, [m]^d \,\|\, M_1 \,\|\, M_{\|L'-p-2r-d}$.



[a] Rigorously, we should write $\tilde{y} \xleftarrow{M_{\text{Link}}} \bar{y}$.

There are two main differences between the attack on the Zipper hash and the second-preimage attack on concatenation combiners (Attack 4) and the preimage attacks on XOR combiners (Attacks 2 and 3). One is that linking $\tilde{x}$ to $\bar{x}$ and $\tilde{y}$ to $\bar{y}$ can be carried out independently, resulting in a meet-in-the-middle-like effect. The other is that the message length is embedded inside the expandable message $\mathcal{M}_{\texttt{SEM}}$, which enables us to choose the length of second preimage to optimize the complexity.

7.2 Details of the Second-Preimage Attack on the Zipper Hash

In this subsection, we present the detailed procedure of Attack 5.

Detailed Steps of Attack 5

1. Fix an arbitrary single-block message value m and get $f_1(\cdot) = h_1(\cdot, m)$ and $f_2(\cdot) = h_2(\cdot, m)$.
   (a) Run the cycle search algorithm several times to locate the largest cycles $\mathcal{C}_1$ and $\mathcal{C}_2$ in $\mathcal{FG}_{f_1}$ and $\mathcal{FG}_{f_2}$, get the cycle lengths L_1 and L_2. Without loss of generality, assume $L_1 \leq L_2$.
   (b) Pick a cyclic node $\bar{x}$ located in $\mathcal{C}_1$ and a cyclic node $\bar{y}$ located in $\mathcal{C}_2$.
   (c) Compute the set of correctable distance bias $\mathcal{D} = \{i \cdot \Delta L \mod L_1 \mid i = 0, 1, \ldots, \#C\}$ as in Step 2 of Attack 3.
2. Run Alg. 5 with a parameter t to develop 2^t nodes in $\mathcal{FG}_{f_1}$ (resp. $\mathcal{FG}_{f_2}$), compute their distance from the target node $\bar{x}$ (resp. $\bar{y}$). Store these nodes in $\mathcal{FG}_{f_1}$ (resp. $\mathcal{FG}_{f_2}$) in a data structure $\mathcal{G}_1$ (resp. $\mathcal{G}_2$).
3. Build a 2^r-Joux's multi-collision $\mathcal{M}_{\texttt{MC1}}$ (resp. $\mathcal{M}_{\texttt{MC2}}$) starting from the cyclic node $\bar{x}$ (resp. $\bar{y}$) and denote its final endpoint by $\hat{x}$ (resp. $\hat{y}$).
4. Construct a simultaneous expandable message $\mathcal{M}_{\texttt{SEM}}$ across the two hash functions that starts from the state $\hat{x}$ in the first pass, and denote its final endpoint by $\tilde{y}$ in the second pass. The details of constructing such an expandable message are provided in Sect. 7.3.
5. Find a single-block $\bar{m}$ mapping $\hat{y}$ to some internal state b_p in the second pass of the original message $\boldsymbol{M}$. The search procedure is trivial and hence omitted. Then, use $\bar{m}$ updating the corresponding chaining state a_p in the first pass to a state $\tilde{x}$, *i.e.*, compute $\tilde{x} = h_1(a_p, \bar{m})$.
6. For each message M_2' in $\mathcal{M}_{\texttt{MC2}}$,
   (a) Compute $\check{x} = h_1^*(\tilde{x}, M_2')$.
   (b) Compute a chain $\vec{x}$ by applying f_1 to update $\check{x}$ iteratively until up to a maximum length 2^{n-t} or until it hits $\mathcal{G}_1$. In the latter case, compute the distance d_1 of $\check{x}$ to $\bar{x}$, and store (d_1, M_2') in a table $\mathcal{T}_1$.
7. For each message M_1' in $\mathcal{M}_{\texttt{MC1}}$,

(a) Compute $\check{y} = h_2^*(\tilde{y}, M_1')$;
(b) Compute a chain $\vec{y}$ by applying f_2 to update $\check{y}$ up to a maximum length 2^{n-t} or until it hits $\mathcal{G}_2$. In the latter case, compute the distance d_2 of $\check{y}$ to $\bar{y}$, and store (d_2, M_1') in a table $\mathcal{T}_2$.

8. Find (d_1, M_2') in $\mathcal{T}_1$ and (d_2, M_1') in $\mathcal{T}_2$ such that $(d_1 - d_2 \mod L_1) \in \mathcal{D}$. The search is a meet-in-the-middle procedure to match elements between $\mathcal{T}_1$ and $\mathcal{T}_2$. Denote the common distance corrected by values in $\mathcal{D}$ by $d \triangleq d_1 + i \cdot L_1 = d_2 + j \cdot L_2$ for some i and j, and the corresponding messages $M_1 \triangleq M_1'$ and $M_2 \triangleq M_2'$ (retrieved from $\mathcal{T}_1$ and $\mathcal{T}_2$).
9. Derive a message $M_{\|q}$ with a block length $q = L' - p - 1 - r - d - r$ from $\mathcal{M}_{\texttt{SEM}}$, where L' is the length of the constructed second-preimage. Construct a message $\boldsymbol{M'} = m_1 \,\|\, m_2 \,\|\, \cdots \,\|\, m_p \,\|\, \bar{m} \,\|\, M_2 \,\|\, [m]^d \,\|\, M_1 \,\|\, M_{\|q}$ and output $\boldsymbol{M'}$ as a second-preimage.

Complexity Analysis. The complexities of each step in this attack are as follows (ignoring constant factors and the factor n):

Step 1:	$2^{n/2}$	**Step 2:**	2^t	**Step 3:**	$2^{n/2}$
Step 4:	$2^{\ell'} + 2^{n/2+2\log_2(n)+1}$	**Step 5:**	$2^{n-\ell}$	**Step 6:**	$2^r \cdot 2^{n-t}$
Step 7:	$2^r \cdot 2^{n-t}$	**Step 8:**	2^r	**Step 9:**	$2^{\ell'}$

The sum of dominant terms is

$$2^t + 2^{\ell'} + 2^{n-\ell} + 2^r \cdot 2^{n-t},$$

where 2^t is the complexity for developing more nodes in the look-ahead procedure; $2^{\ell'}$ is the complexity for building the simultaneous expandable message; $2^r \cdot 2^{n-t}$ is the complexity for generating tables $\mathcal{T}_1$ and $\mathcal{T}_2$.

We first balance the first term with the fourth term by setting $t = r + n - t$, which gives $t = n/2 + r/2$. As a result, the sum of dominant terms is

$$2^{n/2+r/2} + 2^{\ell'} + 2^{n-\ell}.$$

- When the allowed length L' of the second preimage is limited by $2^{n/2}$, we set $\ell' = n/2$ to optimize the complexity. The multi-cycle technique is not applicable. The required number of samplings on pairs of starting nodes before finding one pair reaching any one of the 2^{2r} pairs of target nodes at a common distance is $2^{2n-3n/2}$. Thus, it is required that $2^{2r} = 2^{2n-3n/2} = 2^{n/2}$. This gives $r = n/4$. The total complexity is then $2^{5n/8}$ for all $\ell \geq 3n/8$.
- When the allowed length L' of the second preimage is not limited and can be greater than $2^{n/2}$, multi-cycles can be used. In this case, the required number of samplings on pairs of starting nodes before finding one pair reaching any one of the 2^{2r} pairs of target nodes at a common distance is $2^{2n-3n/2-(\ell'-n/2)}$. Thus, $2^{2r} = 2^{n-\ell'}$. This gives $r = n/2 - \ell'/2$. The total complexity is then $2^{3n/4-\ell'/4} + 2^{\ell'} + 2^{n-\ell}$. We then balance the first two terms by setting $3n/4 - \ell'/4 = \ell'$, which gives $\ell' = 3n/5$.

- For the case $\ell < 2n/5$, the total complexity is $2^{n-\ell}$;
- For the case $2n/5 \leq \ell$, the total complexity is stabilized at $2^{3n/5}$.

(see Fig. 13 for trade-off curves).

7.3 Step 4: Constructing an Expandable Message

The constructing method is similar to that in Sect. 2.6, with slight modifications. Detailed steps and the method used are shown in Alg. 6, where $C \approx n/2+\log(n)$:

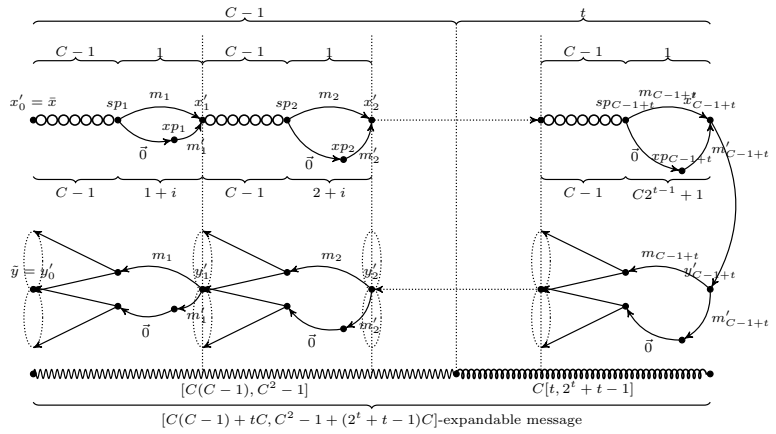
Algorithm 6: Building and using a simultaneous expandable message applicable to attacks on the Zipper hash

1. $x'_0 \leftarrow \hat{x}$
2. For $i \leftarrow 1,\ 2, \cdots, C-1+t$:
   (a) Build a 2^{C-1} standard Joux's multi-collision in h_1 starting from x'_{i-1}, and denote its final endpoint by sp_i.
   (b) Compute $xp_i = h_1^*(sp_i, [0]^s)$, where $[0]^s$ is an all-zero message of size s blocks, where $s = i$ if $i \leq C-1$ and $s = C2^{i-(C-1)-1}$ if $C-1 < i \leq C-1+t$.
   (c) Find a collision $h_1(sp_i, m_i) = h_1(xp_i, m'_i)$ with single block messages $m_i,\ m'_i$. Denote the collision by x'_i.
   (d) We obtain a multi-collision in h_1 with 2^C messages that map x'_{i-1} to x'_i.
      i. Out of these messages, 2^{C-1} are of length b (obtained by combining one of the 2^{C-1} Joux's multi-collisions with m_i). We denote this set of messages by $\mathcal{M}_{\texttt{short}_i}$, where $b = C$.
      ii. Out of these messages, 2^{C-1} are of length b (obtained by combining one of the 2^{C-1} Joux's multi-collisions with $[0]^s \parallel m'_i$) and we denote this set of messages by $\mathcal{M}_{\texttt{long}_i}$, where $b = C+i$ if $i \leq C-1$ and $b = C(2^{i-(C-1)-1}+1)$ if $C-1 < i \leq C-1+t$.
3. Denote the last collision state x'_{C-1+t} by $\ddot{x}$, and compute $\ddot{y} = h_2(h_1(\ddot{x}, m'_{L'}), m'_{L'})$, where $m'_{L'}$ is a message block padded with the length L' of the second preimage.
4. $y'_{C-1+t} \leftarrow \ddot{y}$, $\mathcal{M}_{\texttt{SEMshort}} \leftarrow \emptyset$, $\mathcal{M}_{\texttt{SEMlong}} \leftarrow \emptyset$.
5. For $i \leftarrow C-1+t,\ C-1+t-1,\ \ldots,\ 2,\ 1$:
   (a) For each $\vec{ms}_i \in \mathcal{M}_{\texttt{short}_i}$, compute $u_i = h_2^*(y'_i, \overleftarrow{\vec{ms}_i})$ where $\vec{ms}_i = ms_{i,1} \parallel ms_{i,2} \parallel \cdots \parallel ms_{i,C-1} \parallel ms_{i,C}$ and $\overleftarrow{\vec{ms}_i} = ms_{i,C} \parallel ms_{i,C-1} \parallel \cdots \parallel ms_{i,1}$. Store each pair $(u_i,\ \vec{ms}_i)$ in a table $\mathcal{U}_i$ indexed by u_i. The final size of $\mathcal{U}_i$ is 2^{C-1}.
   (b) For each $\vec{ml}_i \in \mathcal{M}_{\texttt{long}_i}$, compute $v_i = h_2^*(y'_i, \overleftarrow{\vec{ml}_i})$ where $\vec{ml}_i = ml_{i,1} \parallel ml_{i,2} \parallel \cdots \parallel ml_{i,s-1} \parallel ml_{i,s}$ and $\overleftarrow{\vec{ml}_i} = ml_{i,s} \parallel ml_{i,s-1} \parallel \cdots \parallel ml_{i,1}$, where $s = C(2^{i-(C-1)-1}+1)$ if $C-1 < i \leq C-1+t$ and $s = C+i$ if $1 \leq i \leq C-1$. Store each pair $(v_i,\ \vec{ml}_i)$ in a table $\mathcal{V}_i$ indexed by v_i. The final size of $\mathcal{V}_i$ is 2^{C-1}.

(c) Find a match $u_i = v_i$ between $\mathcal{U}_i$ and $\mathcal{V}_i$, and denote the matched state by $y'_{i-1} \triangleq u_i = v_i$. Combine the corresponding message fragment $\vec{ms}_i$ indexed by y'_i with $\mathcal{M}_{\texttt{SEMshort}}$ and $\vec{ml}_i$ indexed by y'_i with $\mathcal{M}_{\texttt{SEMlong}}$, *i.e.*, $\mathcal{M}_{\texttt{SEMshort}} = \vec{ms}_i \parallel \mathcal{M}_{\texttt{SEMshort}}$ and $\mathcal{M}_{\texttt{SEMlong}} = \vec{ml}_i \parallel \mathcal{M}_{\texttt{SEMlong}}$.

Then, the whole simultaneous expandable message $\mathcal{M}_{\texttt{SEM}}$ can be fully defined by $\mathcal{M}_{\texttt{SEMshort}}$ and $\mathcal{M}_{\texttt{SEMlong}}$. For any length κ lying in the appropriate range of $[C(C-1)+tC,\ C^2-1+C(2^t+t-1)]$, one can construct a message $M_{\|\kappa}$ mapping $\hat{x}$ to $\tilde{y} = y'_0$ through h_1 and h_2 by picking messages fragment either from $\mathcal{M}_{\texttt{SEMshort}}$ or from $\mathcal{M}_{\texttt{SEMlong}}$ as also described in Sect.2.6:

1. Select the length $\kappa' \in [C(C-1), C^2-1]$ such that $\kappa' = \kappa \bmod C$, defining the first $C-1$ message fragment choices: selecting the message fragment $\vec{ms}_i$ in $\mathcal{M}_{\texttt{SEMshort}}$ for $1 \le i \le C-1$ and $i \ne \kappa' - C$; selecting the message fragment $\vec{ml}_i$ in $\mathcal{M}_{\texttt{SEMlong}}$ for $i = \kappa' - C$.
2. Compute $kp \leftarrow (\kappa - \kappa')/C$, which is an integer in the range of $[t, 2^t + t - 1]$, and select the final t message fragment choices as in a standard expandable message using the binary representation of $kp - t$.



8 Second-Preimage Attack on Hash-Twice

In this section, we present an efficient second-preimage attack on another cascade hash construction – Hash-Twice (a generalized specification $\mathcal{HT}(M) \triangleq \mathcal{H}_2(\mathcal{H}_1(IV, M), M)$). Similar to the previous second-preimage attack on Hash-Twice in [ABDK09], this attack builds a diamond structure for one hash function by exploiting messages in a long multi-collision built for the other hash function. Like all our previous functional-graph-based (deep-iterates-based and multi-cycles-based) attacks, it improves the attack from [ABDK09] because of the efficiency brought by exploiting the special nodes in the functional graphs. It follows the same structure as the second-preimage attack on the concatenation

combiner, but the result shows that attacking Hash-Twice can be much more efficient than attacking the concatenation combiner. That is mainly because the attack tries to connect to an n-bit internal state in the case of Hash-Twice, instead of a $2n$-bit internal state in the case of the concatenation combiner. Note that similar to all previous functional-graph-based attacks, this attack applies when the underlying hash functions use the MD construction.

8.1 Overview of the Attack

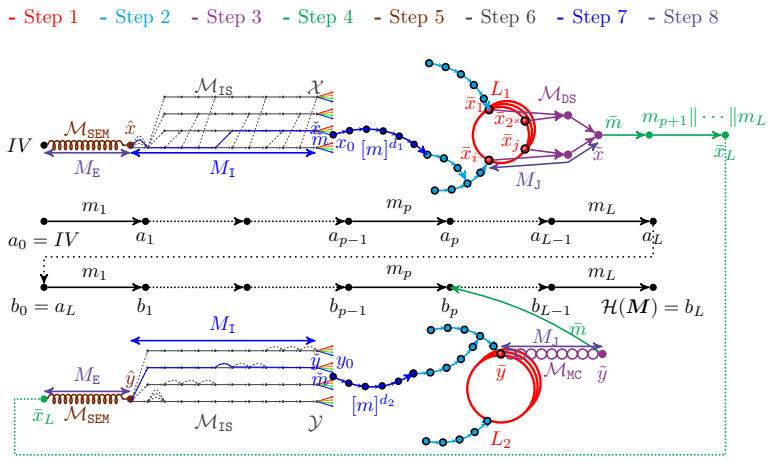
Given a challenge message $\boldsymbol{M} = m_1 \parallel m_2 \parallel \cdots \parallel m_L$, the goal of the second-preimage attack on Hash-Twice is to find another message $\boldsymbol{M'}$ such that $\mathcal{H}_2(\mathcal{H}_1(IV, \boldsymbol{M'}), \boldsymbol{M'}) = \mathcal{H}_2(\mathcal{H}_1(IV, \boldsymbol{M}), \boldsymbol{M})$. The framework of this attack is sketched as follows.

Attack 6: Second-preimage attack on Hash-Twice

- **Phase 1:** Generate a set of pairs of nodes $\mathcal{S} = \{(\bar{x}_1, \bar{y}), (\bar{x}_2, \bar{y}), \ldots, (\bar{x}_{2^s}, \bar{y})\}$, where $\bar{x}_i$'s are cyclic nodes randomly located in the largest cycle of $\mathcal{FG}_{f_1}$, and $\bar{y}$ is a cyclic node in the largest cycle of $\mathcal{FG}_{f_2}$.
  In addition, like in Step 2 of Attack 3, compute the set of correctable distance bias $\mathcal{D} = \{j \cdot \Delta L \mod L_1 \mid j = 0, 1, \ldots, \#C\}$ with parameters L_1, L_2 and $\#C$ (where $\Delta L = L_2 \mod L_1$, L_1 and L_2 are the two cycle lengths, $\#C$ is the number of cycles to the maximum).
- **Phase 2:** Build a long Joux's multi-collision $\mathcal{M}_{\texttt{MC}}$ starting from $\bar{y}$ and denote its endpoint by $\tilde{y}$. Then, by exploiting messages in $\mathcal{M}_{\texttt{MC}}$, build a diamond structure $\mathcal{M}_{\texttt{DS}}$ with the cyclic nodes $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_{2^s}$ as leaves and denote its root by $\tilde{x}$. Find a message block $\bar{m}$ mapping the endpoint $\tilde{y}$ of Joux's multi-collision to one of the chaining states b_p in the second pass of the original message. After finding $\bar{m}$, starting from the root $\tilde{x}$ of the diamond $\mathcal{M}_{\texttt{DS}}$, compute the final state $\bar{x}_L$ in the first pass with message fragment $\bar{m} \parallel m_{p+1} \parallel \cdots \parallel m_L$.
- **Phase 3:** Build a simultaneous expandable message $\mathcal{M}_{\texttt{SEM}}$ (with lengths covering roughly $[n^2, L]$) starting from $(IV, \bar{x}_L)$. Denote its endpoints by $(\hat{x}, \hat{y})$.
- **Phase 4:** Find a message fragment $M_{\texttt{Link}}$ such that $(\hat{x}, \hat{y}) \xrightarrow{M_{\texttt{Link}}} (\bar{x}, \bar{y})$ for some $(\bar{x}, \bar{y}) \in \mathcal{S}$.
  This is done by first build a 2^r-interchange structure $\mathcal{M}_{\texttt{IS}}$ starting from $(\hat{x}, \hat{y})$. Denote its two sets of endpoints by $\mathcal{X}$ and $\mathcal{Y}$. Then, for states in $\mathcal{X}$ and $\mathcal{Y}$, launch a meet-in-the-middle procedure to find a pair of nodes (x_0, y_0) with $x_0 = h_1(\check{x}, \check{m})$ and $y_0 = h_1(\check{y}, \check{m})$ and $\check{x} \in \mathcal{X}$ and $\check{y} \in \mathcal{Y}$, such that in $\mathcal{FG}_{f_1}$ and $\mathcal{FG}_{f_2}$, the distance difference of x_0 and y_0 from some targeted nodes $(\bar{x}, \bar{y}) \in \mathcal{S}$ is correctable by the values in $\mathcal{D}$. Denote the common distance after corrected as d. Retrieve the message fragment $M_{\texttt{I}}$ from $\mathcal{M}_{\texttt{IS}}$ such that $(\hat{x}, \hat{y}) \xrightarrow{M_{\texttt{I}}} (\check{x}, \check{y})$. The desired $M_{\texttt{Link}}$ is then defined as $M_{\texttt{Link}} \triangleq M_{\texttt{I}} \parallel \check{m} \parallel [m]^d$.

At the end, select a message fragment $M_{\mathtt{E}} \in \mathcal{M}_{\mathtt{SEM}}$ with a proper block length, and a message fragment $M_{\mathtt{J}} \in \mathcal{M}_{\mathtt{DS}}$ such that $\bar{x} \xrightarrow{M_{\mathtt{J}}} \tilde{x}$, and construct a second preimage:

$$\begin{aligned} \boldsymbol{M}' &\triangleq M_{\mathtt{E}} \parallel M_{\mathtt{Link}} \parallel M_{\mathtt{J}} \parallel \bar{m} \parallel m_{p+1} \parallel \cdots \parallel m_L \\ &= M_{\mathtt{E}} \parallel M_{\mathtt{I}} \parallel \check{m} \parallel [m]^d \parallel M_{\mathtt{J}} \parallel \bar{m} \parallel m_{p+1} \parallel \cdots \parallel m_L. \end{aligned}$$



8.2 Details of the Second-Preimage Attack on Hash-Twice Based on Multi-Cycles, Diamond and Interchange Structure for Long Messages

The detailed steps of the second-preimage attack on Hash-Twice are follows:

Detailed Steps of Attack 6

1. Fix an arbitrary single-block message m and construct $f_1(\cdot) \triangleq h_1(\cdot, m)$ and $f_2(\cdot) \triangleq h_2(\cdot, m)$.
   (a) Run the cycle search algorithm several times to locate the largest cycles in $\mathcal{FG}_{f_1}$ and $\mathcal{FG}_{f_2}$, and obtain the cycle lengths L_1 and L_2. Without loss of generality, assume $L_1 \leq L_2$.
   (b) Generate a set of 2^s pairs of target nodes

$$\mathcal{S} = \{(\bar{x}_1, \bar{y}), (\bar{x}_2, \bar{y}), \ldots, (\bar{x}_{2^s}, \bar{y})\},$$

   where $\bar{x}_i$ are cyclic nodes randomly located in the largest cycle of $\mathcal{FG}_{f_1}$ and $\bar{y}$ is a cyclic node in the largest cycle of $\mathcal{FG}_{f_2}$.
   (c) Compute the set of correctable distance bias

$$\mathcal{D} = \{j \cdot \Delta L \mod L_1 \mid j = 0, 1, \ldots, \#C\}$$

with parameters L_1, L_2 and $\#C$ (where $\#C$ is the allowed maximum number of cycles and $\Delta L = L_2 \mod L_1$), as in Step 2 of Attack 3.

2. Develop 2^t nodes in $\mathcal{FG}_{f_1}$ (resp. $\mathcal{FG}_{f_2}$) by running Alg. 5, and record these together with their distances from a particular target node $\bar{x}_i$ (resp. $\bar{y}$) in $\mathcal{FG}_{f_1}$ (resp. $\mathcal{FG}_{f_2}$) in a table $\mathcal{T}_x$ (resp. $\mathcal{T}_y$). Note that the distance of a node from all other target nodes $\bar{x}_j$ can be directly deduced from the distance of it from the particular target node $\bar{x}_i$, as explained in Step 4c of Attack 3.
3. Build a long Joux's multi-collision $\mathcal{M}_{\texttt{MC}}$ of length $s \cdot n/2$ (blocks) starting from state $\bar{y}$, and denote its endpoint by $\tilde{y}$. Then, by exploiting messages in $\mathcal{M}_{\texttt{MC}}$, build a diamond structure $\mathcal{M}_{\texttt{DS}}$ starting from cyclic nodes $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_{2^s}$ (as leaves) and denote its root by $\tilde{x}$.
4. Find a message block $\bar{m}$ mapping the endpoint $\tilde{y}$ of Joux's multi-collision to one of the chaining values b_p in the second pass of the original message, as done in the second-preimage attack on a single MD [KS05]. After finding $\bar{m}$, starting from the root $\tilde{x}$ of the diamond, directly compute a final state (denoted by $\bar{x}_L$) in the first pass with message fragment $\bar{m} \parallel m_{p+1} \parallel \cdots \parallel m_L$. The initial state in the second pass on the second preimage is then determined to be $\bar{x}_L$.
5. Build a parallel simultaneous expandable message $\mathcal{M}_{\texttt{SEM}}$ starting from $(IV, \bar{x}_L)$, and denote their endpoints by $(\hat{x}, \hat{y})$.
6. Build a 2^r-interchange structure $\mathcal{M}_{\texttt{IS}}$ starting from $(\hat{x}, \hat{y})$, and denote its two sets of endpoints by $\mathcal{X}$ and $\mathcal{Y}$.
7. Find a message fragment $M_{\texttt{Link}}$ such that $(\hat{x}, \hat{y}) \xrightarrow{M_{\texttt{Link}}} (\bar{x}, \bar{y})$ for some $(\bar{x}, \bar{y}) \in \mathcal{S}$. The search procedure is described as follows.
   (a) Select a random single-block message w.
   (b) Initialize a table $\mathcal{T}_y$ as empty.
   (c) For each of the 2^r states $\check{y}' \in \mathcal{Y}$:
      – Compute $y_0 = h_2(\check{y}', w)$
      – Derive the distance $d_{\bar{y}}$ of y_0 from the target node $\bar{y}$ in $\mathcal{FG}_{f_2}$ (if it is undefined $\perp$, go to Step 7a), store $(\check{y}', d_{\bar{y}})$ in $\mathcal{T}_y$.
   (d) For each of the 2^r states $\check{x}' \in \mathcal{X}$:
      – Compute $x_0 = h_1(\check{x}', w)$
      – For each of the 2^s target node $\bar{x}_i \in \{\bar{x}_1, \bar{x}_2, \cdots, \bar{x}_{2^s}\}$
         • Derive the distance of x_0 from $\bar{x}_i$ (as described in Step 4c of Attack 3), denote it by $d_{\bar{x}_i}$.
         • Make a match between $d_{\bar{x}_i}$ and elements in $\mathcal{T}_y$ by checking whether there exists a $d_{\bar{y}} \in \mathcal{T}_y$ such that $(d_{\bar{x}_i} - d_{\bar{y}} \mod L_1) \in \mathcal{D}$. If one exists, retrieve $\check{y}'$ corresponding to $d_{\bar{y}}$ from $\mathcal{T}_y$. Derive the common distance $d \triangleq d_{\bar{x}_i} + j \cdot L_1 = d_{\bar{y}} + k \cdot L_2$. Set $\check{m} \triangleq w$, $\check{x} \triangleq \check{x}'$, $\check{y} \triangleq \check{y}'$, and $\bar{x} \triangleq \bar{x}_i$. Retrieve the message fragment $M_{\texttt{I}}$ from $\mathcal{M}_{\texttt{IS}}$ such that $(\hat{x}, \hat{y}) \xrightarrow{M_{\texttt{I}}} (\check{x}, \check{y})$. Set $M_{\texttt{Link}} \triangleq M_{\texttt{I}} \parallel \check{m} \parallel [m]^d$, and go to the next step.

8. Select a message fragment $M_{\mathtt{E}} \in \mathcal{M}_{\mathtt{SEM}}$ with a proper length $q = p - (2^{t+1} + (2^t - 1)^2 \cdot n/2) - 1 - d - s \cdot n/2 - 1 = p - 2^{t+1} - (2^t - 1)^2 \cdot n/2 - d - s \cdot n/2 - 2$, select the message fragment $M_{\mathtt{J}} \in \mathcal{M}_{\mathtt{DS}}$ such that $\bar{x} \xrightarrow{M_{\mathtt{J}}} \tilde{x}$, and construct a second preimage:

$$\begin{aligned} \boldsymbol{M'} &\triangleq M_{\mathtt{E}} \parallel M_{\mathtt{Link}} \parallel M_{\mathtt{J}} \parallel \bar{m} \parallel m_{p+1} \parallel \cdots \parallel m_L \\ &= M_{\mathtt{E}} \parallel M_{\mathtt{I}} \parallel \check{m} \parallel [m]^d \parallel M_{\mathtt{J}} \parallel \bar{m} \parallel m_{p+1} \parallel \cdots \parallel m_L. \end{aligned}$$

Complexity Analysis. The complexity of each step in this second-preimage attack on Hash-Twice is as follows (denote $L = 2^\ell$):

Step 1: $2^{n/2}$ **Step 2:** 2^t **Step 3:** $n\sqrt{s} \cdot 2^{n/2+s/2}$

Step 4: $2^{n-\ell} + 2^\ell$ **Step 5:** $2^\ell + n^2 \cdot 2^{n/2}$ **Step 6:** $2^{n/2+2r}$

Step 7: $2^r \cdot 2^{n-t} \cdot 2^{n-2r-s-\ell}$ **Step 8:** 2^ℓ

For $\ell > n/2$, we have $2^{n-\ell} < 2^{n/2}$. The sum of dominant terms is (ignoring polynomial factors)

$$2^t + 2^{n/2+s/2} + 2^\ell + 2^{n/2+2r} + 2^{2n-t-r-s-\ell}.$$

We balance different terms by setting $t = n/2 + 2r = n/2 + s/2 = 2n - t - r - s - \ell$, *i.e.*, $r = t/2 - n/4$, $s = 2t - n$ and $t = 13n/18 - 2\ell/9$. Consequently, the total complexity is approximately

$$2^\ell + 2^{13n/18-2\ell/9}.$$

The improved attack is valid for all $\ell > n/2$ (even when we account for the message length 2^{2r} of the interchange structure, which should be less the 2^ℓ). The optimal complexity for this attack is $2^{13n/22}$, obtained when $2^\ell = 2^{13n/22}$. Compared with the original optimal complexity $2^{2n/3}$ for messages of length $2^{13n/22}$, the improvement is $2^{5n/66}$ (see Fig. 13 for a trade-off curve).

8.3 Details of the Second-Preimage Attack on Hash-Twice Based on Deep-iterates, Diamond and Interchange Structure for Short Messages

Note that for $L = 2^\ell \leq 2^{n/2}$, we can no longer apply the multi-cycles technique. However, we can still choose deep-iterates with depth less than $2^{n/2}$ as target nodes and selected proper iterate depth to improve the second-preimage attack on Hash-Twice for short messages.

The procedure of this new attack is similar to that of the previous attack in Sect.8.2. The difference lies in that in Step 1, we collect a set of 2^s pairs of target nodes $\{(\bar{x}_1, \bar{y}), (\bar{x}_2, \bar{y}), \ldots, (\bar{x}_{2^s}, \bar{y})\}$, where $\bar{x}_i$'s are 2^{n-g}-th iterate nodes in $\mathcal{FG}_{f_1}$ for $1 \leq i \leq 2^s$ and $\bar{y}$ is a 2^{n-g}-th iterate node in $\mathcal{FG}_{f_2}$. It is required

that $g \geq \max(n/2, n-\ell)$. In addition, in Steps 2 and 7, we use the distinguish-point technique in the look-ahead procedure as we did in the preimage attack on XOR combiner in Sect.4.2. The procedures in other steps are the same as those in Sect.8.2.

In this case, required number of samplings before finding a pair of $(\check{x}, \check{y})$ such that they reach one of a target pair of nodes $(\bar{x}_i, \bar{y})$ at a common distance changes to be $2^{3g-n-s-2r}$. Consequently, the complexity of Step 7 is $2^r \cdot 2^{3g-n-s-2r} \cdot (2^{n-t} + 2^{\ell+t-n} + 2^{\ell+s-g})$. Thus, complexity of each step in this second-preimage attack on Hash-Twice for short message is as follows:

Step 1: 2^g **Step 2:** 2^t **Step 3:** $n\sqrt{s} \cdot 2^{n/2+s/2}$

Step 4: $2^{n-\ell} + 2^{\ell}$ **Step 5:** $2^{\ell} + n^2 \cdot 2^{n/2}$ **Step 6:** $2^{n/2+2r}$

Step 7: $2^{3g-t-s-r} + 2^{3g+t-s-r+\ell-2n} + 2^{2g+\ell-n-r}$ **Step 8:** 2^{ℓ}

For $\ell \leq n/2$, we have $2^{n-\ell} \geq 2^{n/2}$. Thus, the sum of dominant terms is

$$2^g + 2^t + 2^{n/2+s/2} + 2^{n-\ell} + 2^{n/2+2r} + 2^{3g-s-t-r} + 2^{3g+t-s-r+\ell-2n} + 2^{2g+\ell-n-r}.$$

We first set $t = n/2+s/2 = n/2+2r = 3g-s-t-r$, *i.e.*, $t = 2g/3+5n/18$, $s = 4g/3-4n/9$, and $r = g/3-n/9$ to make a balance. The sum of dominant terms is $2^g + 2^{2g/3+5n/18} + 2^{n-\ell} + 2^{2g+\ell-7n/6} + 2^{5g/3+\ell-8n/9}$. We optimize the attack by picking the minimal value of g under the restriction $g \geq \max(n/2, n-\ell)$, *i.e.*, $g = n-\ell$ for $\ell \leq n/2$. Consequently, the total complexity becomes (for $g = n-\ell$ and $\ell < n/2$, with the last two terms always less than $2^{n-\ell}$, the distinguished points method allowed us to resolve with no overhead the complication of keeping track of distances from 2^s target nodes):

$$2^{17n/18-2\ell/3} + 2^{n-\ell}.$$

The improved attack is better than that in [ABDK09] for $\ell > 5n/12$ (in which case $17n/18 - 2\ell/3 < 2n/3$, the message length 2^{2r} of the interchange structure is less the 2^{ℓ}, thus applicable). The optimal complexity for this attack is $2^{11n/18}$, obtained when $\ell = n/2$. Compared with the previous best-known complexity $2^{2n/3}$ at message length $L = 2^{n/2}$, the improvement is $2^{n/18}$ (see Fig. 13 for a trade-off curve).

9 More Applications and Extensions

9.1 Applications Beyond MD Construction and Beyond XOR Operation

Application to HAIFA mode. The first preimage attack on the XOR combiner purely bases on interchange structure. Thus, it works identically if the hash functions use the HAIFA mode rather than the plain Merkle-Damgård iteration, whereas the other attacks all based on functional graphs requiring identical compression functions and thus cannot work if the underlying hash functions use the HAIFA mode.

Application to Cryptophia's short combiner. All of our attacks on XOR combiner can also be applied to Cryptophia's short combiner, as proposed by Mittelbach [Mit13], and to the revised version of Mennink and Preneel [MP14]. This combiner computes the sum of two hash functions with some pre-processing of the message to allow non-independent functions:

$$\begin{aligned} C(M) &= \mathcal{H}_1\left(\tilde{m}_1^1 \parallel \ldots \parallel \tilde{m}_\ell^1\right) \oplus \mathcal{H}_2\left(\tilde{m}_1^2 \parallel \ldots \parallel \tilde{m}_\ell^2\right) \\ \tilde{m}_j^1 &= \mathcal{H}_1(0 \parallel l_1 \parallel m_j \oplus k_1) \oplus \mathcal{H}_2(0 \parallel l_2 \parallel m_j \oplus k_2) \\ \tilde{m}_j^2 &= \mathcal{H}_1(1 \parallel l_1 \parallel m_j \oplus k_1) \oplus \mathcal{H}_2(1 \parallel l_2 \parallel m_j \oplus k_2) \end{aligned}$$

where k_1, k_2, l_1, l_2 is a randomly chosen key. The security proof in the ideal model shows that C is optimally preimage resistant if at least one of the hash functions is ideal.

However, if both $\mathcal{H}_1$ and $\mathcal{H}_2$ are narrow-pipe, we can apply our preimage attacks with the same corresponding complexity. This does not violate the security proof because we need both functions to be narrow-pipe and hence not n-bit ideal[25]. From a practical point of view, though, they show that in many cases (*e.g.* using SHA-512 and Whirlpool) the combiner is *weaker* than the initial functions.

Application beyond XOR. All preimage attacks on XOR combiner can easily be extended to $\mathcal{H}_1(M) \odot \mathcal{H}_2(M)$ where $\odot$ denotes an easy to invert group operation (for instance, a modular addition rather than the exclusive or). These attacks can also be extended to hash functions $\mathcal{H}_1$ and/or $\mathcal{H}_2$ using an internal checksum, such as the GOST family of hash functions, using pairs of blocks with a constant sum.

9.2 Applications to the Combination of Wide-Pipe Hash Functions

Our attacks can also be used when the internal state size n' is (not much) larger than the output size n.

The interchange-structure-based preimage attack on the XOR combiner. In Attack 1, the complexity of building a 2^t-interchange structure is related to n' as $(n'/2) \cdot 2^{2t+n'/2}$. On the other hand, the complexity of the meet-in-the-middle preimage search is related to n as 2^{n-t}. The optimal complexity is $(n'/2) \cdot 2^{2n/3+n'/6}$ by matching the two complexities with $t = n/3 - n'/6$. Therefore, our attack can be applied as long as $n' + 6\log(n') \leq 2n$ holds. For instance, we can compute preimages of SHA-224 $\oplus$ BLAKE-224 using the interchange-structure-based attack with complexity roughly 2^{199}.[26]

[25] A large multi-collisions can be built with a cost of roughly $2^{n/2}$ in a narrow-pipe function, but this costs almost 2^n for an ideal hash function.

[26] However, the message length can be a problem with some hash functions that do not accept long inputs. For example, SHA-256 and SHA-224 are only defined for

The deep-iterates-based preimage attack on the XOR combiner. For parameters $g_1 \geq \max(n'/2,$
$n' - \ell)$ and $s \geq 0$, the complexity of each phase in Attack 2 is as follows:

Phase 1: $2^\ell + {n'}^2 \cdot 2^{n'/2}$ **Phase 2:** 2^{n+s-g_1}

Phase 3: $2^{3g_1/2-s/2} + 2^{\ell+9g_1/2-2n'-3s/2} + 2^{\ell+2g_1-n'}$

We balance the time complexities of different phases similar to what we did before. In case of $\ell \leq n'/2$, we set $g_1 = n' - \ell$, the total time complexity of the attack is $2^{n/3+2n'/3-2\ell/3} + 2^{n'-\ell}$. The optimal complexity is $2^{n/3+n'/3}$,[27] obtained for $\ell = n'/2$. It is less than 2^n when $n' < 2n$.

The multi-cycles-based preimage attack on the XOR combiner. The complexity of each detailed step in Attack 3 is no longer related to the output size n but related to the internal state size n' except for Step 3. For Step 3, the complexity is $2^{s+n-n'/2}$. For other steps, the complexity can be obtained by simply replacing n with n' from the original formula. Then, the overall complexity is approximately (note that $\ell \geq n'/2$)

$$2^\ell + 2^{s+n-n'/2} + 2^t + 2^{2n'-t-s-\ell}.$$

We balance the last three terms by setting $s + n - n'/2 = t = 2n' - t - s - \ell$, *i.e.*, $s = n' - 2n/3 - \ell/3$, $t = n'/2 + n/3 - \ell/3$. Then, the total complexity is $2^\ell + 2^{n'/2+n/3-\ell/3}$. The optimal complexity is $2^{3n'/8+n/4}$ for $\ell = 3n'/8 + n/4$. The complexity is less than 2^n when $n'/2 < \ell < n$ and $n' < 2n$.

The deep-iterates-based second-preimage attack on the concatenation combiner. The complexity of Attack 4 is no longer related to the output size n but rather is related to the internal state size n' and the message length $L = 2^\ell$. The time complexity is $2^{(3/5)\cdot(2n'-\ell)}$ as long as $\ell \leq 3n'/4$ and is less than 2^n for $\ell > 2n' - 5n/3$. Therefore, this attack can be applied when $n' < 4n/3$.

The second-preimage attack on the Zipper hash. The complexity of Attack 5 is no longer related to the output size n but is related to the internal state size n'. We can get the complexity by simply replacing n with n' in the formula. Accordingly, when the length L' of the second preimage is limited by $2^{n'/2}$, the optimal complexity is $2^{5n'/8}$ for all $\ell \geq 3n'/8$. It is less than 2^n when $n' < 8n/5$. When the length L' is not limited, the optimal complexity is $2^{3n'/5}$ for all $\ell \geq 2n'/5$. It is less than 2^n when $n' < 5n/3$.

messages with less than 2^{64} bits (*i.e.*, 2^{55} blocks). In this case, one can apply the attack with a smaller value of t: this reduces the length of the messages at the cost of more time spent in the preimage search step. Thus, to mount a preimage attack against SHA-224 $\oplus$ BLAKE-224, we should use $t = 24$ instead of $t = 32$. Then, the optimal complexity is 2^{200} instead of 2^{199}.

[27] Note that $n/3 + n'/3 > n'/2$ when $n' < 2n$.

The second-preimage attack on Hash-Twice. The complexity of Attack 6 is no longer related to the output size n but rather is related to the internal state size n'. Accordingly, when the length $L = 2^\ell$ is limited by $2^{n'/2}$, the total complexity is $2^{n'-\ell} + 2^{17n'/18-2\ell/3}$. The optimal complexity is $2^{11n'/18}$, obtained for $\ell = n'/2$, which is less than 2^n when $n' < 18n/11$. When the length is not limited, the total complexity is $2^\ell + 2^{13n'/18-2\ell/9}$. The optimal complexity is $2^{13n'/22}$, obtained for $\ell = 13n'/22$, which is less than 2^n when $n' < 22n/13$.

9.3 Extensions to the combination of three or more hash functions

The interchange-structure-based preimage attack on the XOR combiner. The interchange-structure-based attack on the XOR combiner, *i.e.*, Attack 1, can be extended to the sum of three or more hash functions. To attack the sum of k functions, two different strategies are possible: either we use a simpler structure that only gives two degrees of freedom and fixes $k-2$ functions to a constant value, or we build an interchange structure to control all the k functions independently.

Controlling only two functions. The easiest way to extend the attack is to use a single chain in the $k-2$ extra hash functions. The procedure to build a switch is modified in order to use multi-collisions for $k-1$ functions instead a simple multi-collisions for one function; this costs $O(n^{k-1} \cdot 2^{n/2})$ using Joux's method [Jou04]. As in the basic attack, we need $O(t^2)$ switches to generate a 2^t-interchange for two functions, and the preimage search costs $O(2^{n-t})$; the optimal complexity is therefore $O(n^{k-1} \cdot 2^{5n/6})$ with $t = n/6$.

Controlling all the functions. Alternatively, we can build a more complex interchange structure in order to control all the functions independently. When attacking three functions, we will use the switch structure to jump from chains $(\vec{a}_{j_0}, \vec{b}_{k_0}, \vec{c}_{l_0})$ to $(\vec{a}_{j_0}, \vec{b}_{k_0}, \vec{c}_{l_1})$ (or $(\vec{a}_{j_0}, \vec{b}_{k_1}, \vec{c}_{l_0})$ or $(\vec{a}_{j_1}, \vec{b}_{k_0}, \vec{c}_{l_0})$). We need $2^{3t} - 1$ switches in the interchange structure to reach all the 2^{3t} triplets of chains (a switch makes only one new triplet reachable). Each switch is built using a $2^{n/2}$-multi-collision on two functions, which can be built for a cost of $O(n^2 \cdot 2^{n/2})$ following Joux's technique [Jou04]. Therefore, we can build a 2^t-interchange for a cost of $O(n^2 \cdot 2^{3t+n/2})$. More generally, for the sum of k hash functions, we can build an interchange structure for k functions for a cost of $O(n^{k-1} \cdot 2^{kt+n/2})$.

In the preimage search phase, we generate k lists of size 2^t, and we need to detect efficiently whether we can combine then to generate a zero-sum. This problem can be solved using an algorithm similar to Wagner's generalized birthday algorithm [Wag02]. If $k = 2^\kappa$, we find a solution with probability $O(2^{(\kappa+1)\cdot t-n})$ for a cost of $O(k \cdot 2^t)$. Therefore, the preimage search costs $O(k \cdot 2^{n-\kappa t})$. With $k = 4$ (*i.e.*, $\kappa = 2$), this yields a complexity of $O(n^3 \cdot 2^{5n/6})$. However, this approach is less efficient than the previous one for $k = 3$ and for $k > 4$.

To summarize, attacking the sum of k hash functions ($k \geq 2$) using interchange structure costs $O(n^{k-1} \cdot 2^{5n/6})$. Controlling chains independently in more than

two hash function might be useful for further work, but it does not improve the preimage attack on the sum of k hash functions.

The deep-iterates-based preimage attack on the XOR combiner. Suppose the combiner outputs the sum of k hash digests of n bits. To extend Attack 2, one needs to first construct a simultaneous expandable message for k independent hash functions, this costs $O(2^\ell + n^{2(k-1)} \cdot 2^{n/2})$ by setting $C \approx (n/2 + \log(n))^{k-1}$ (see Fig.9a).

To extend Phase 2, one collects 2^{g_1} deep-iterates for each of the k random functional graphs. Then, tries to find a set of 2^s tuples of k deep-iterates by mapping a single-block message to states whose sum equals the target V. Again, this can be solved using an algorithm similar to Wagner's generalized birthday algorithm, with k lists of size 2^{g_1}. Let $k = 2^\kappa$, and then finding 2^s tuples costs $O(2^s \cdot k \cdot 2^{n-\kappa g_1}) \approx 2^{n+s-\kappa g_1}$.

To extend Phase 3, one expands the k functional graphs independently with parameter g_2 in the look-ahead procedure and then tries to find a tuple of k starting nodes simultaneously hitting one of the 2^s tuples of k deep-iterates obtained in Phase 2. Note that for one tuple of k deep-iterates of depth 2^d in independent random functional graphs, the probability for a tuple of k random nodes reaching them simultaneously is approximately $2^{-kn} \sum_{i=1}^{2^d} i^k \approx 2^{(k+1)d-kn}$. In the attack, $d = n - g_1$. Thus, the complexity of Phase 3 becomes

$$\begin{aligned} &2^{g_2} + 2^{kn-(k+1)(n-g_1)-s} \cdot (2^{n-g_2} + 2^{l+g_2-n} + 2^{l+s-g_1}) \\ =&2^{g_2} + 2^{(k+1)g_1-g_2-s} + 2^{(k+1)g_1+g_2+l-2n-s} + 2^{kg_1+l-n} \end{aligned}$$

Note that in this attack, the restrictions are $\ell \leq n/2$ and $g_1 \geq \max(n/2, n-l)$. Thus, the last term in the complexity of Phase 3, *i.e.*, 2^{kg_1+l-n}, cannot be less than 2^n for $k \geq 3$. In other words, when combining more than two hashes, under any configuration for parameters of the attack, the distinguished points method no longer allowed us to resolve with no overhead the complication of keeping track of distances from the $\mathcal{S}$-nodes. Thus, this extended attack is not more efficient than 2^n.

The multi-cycles-based preimage attack on the XOR combiner. The extension of Attack 3 is very similar to the above extension of the deep-iterates-based preimage attack. However, because this attack does not need to use the distinguished points method, it can be more efficient than 2^n for $k \geq 3$. Let $k = 2^\kappa$. Following a similar analysis for the complexity of each step to the above one, we have the following:

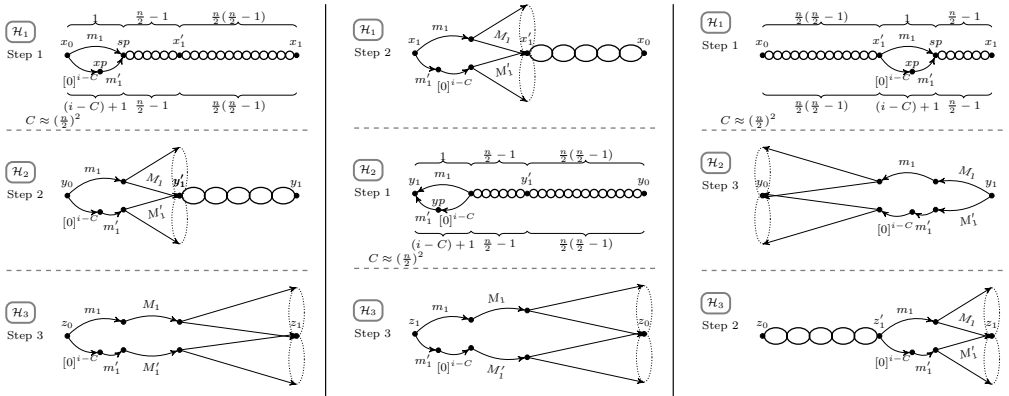
Step 1: $2^\ell + n^{2(k-1)} \cdot 2^{n/2}$	**Step 2:** $2^{n/2} + 2^{\ell-n/2}$	**Step 3:** $2^{s+n-\kappa \cdot n/2}$
Step 4: $k \cdot 2^t$	**Step 5:** $2^{n-t+kn/2-s-\ell}$	**Step 6:** 2^ℓ

We make a balance by setting $s + n - \kappa \cdot n/2 = t = n - t + kn/2 - s - \ell$, *i.e.*, $s = (k+2\kappa-2)n/6 - \ell/3$, and $t = (k-\kappa)n/6 + 2n/3 - \ell/3$. Then, the complexity

becomes $2^{\ell} + 2^{(k-\kappa)n/6+2n/3-\ell/3}$. It is optimally $2^{(k-\kappa+4)n/8}$ by setting $\ell = (k - \kappa + 4)n/8$ (in this attack, we assume $n/2 < \ell < n$).

Accordingly, it costs less than 2^n computations for $k < 7$. For $k = 3$, the optimal complexity is approximately $2^{(7-\log_2 3)n/8} \approx 2^{0.677n}$. For $k = 4$, the optimal complexity is approximately $2^{3n/4} \approx 2^{0.75n}$. For $k = 5$ and $k = 6$, this extension is less efficient than the above extended interchange-structure-based attack.

The deep-iterates-based second-preimage attack on the concatenation combiner. Suppose the combiner outputs the concatenation of k hash digests of n bits. The direct way to extend Attack 4 is to simultaneously control k hash functions. To do that, again, one needs to first construct a simultaneous expandable message for k independent hash functions, this costs $O(2^{\ell} + n^{2(k-1)} \cdot 2^{n/2})$ (see Fig.9a). To extend Phase 2, one collects 2^{g_1} deep iterates for each of the k random functional graphs and then tries to find a tuple of k deep iterates hitting k internal states at the same offset p which uniformly distributed in the interval $[1, 2^{\ell}]$. The complexity of Phase 2 is $2^{g_1} \cdot 2^{-\ell} \cdot 2^{kn-kg_1} = 2^{kn-(k-1)g_1-\ell}$. To extend Phase 3, one expands the k functional graphs with parameter g_2 independently and tries to find a tuple of k starting nodes simultaneously hitting the tuple of k deep iterates obtained in Phase 2. As calculated above, for a tuple of k deep iterates of depth 2^{n-g_1} in independent random functional graphs, the probability for a tuple of k random nodes reaching them simultaneously is about $2^{(k+1)(n-g_1)-kn}$. Thus, the complexity of Phase 3 becomes $2^{(k+1)g_1/2}$ by setting $g_2 = (n - g_2) + kn - (k + 1)(n - g_1)$, *i.e.*, $g_2 = (k + 1)g_1/2$. After making a balance between different phases by setting $g_1 = 2(kn - \ell)/(3k - 1)$, one will find that the optimal complexity is $2^{(k+1)n/4}$, obtained for $l = (k+1)n/4$. Thus, for $k \geq 3$, the attack is not more efficient than 2^n.



(a) Parallel (b) Zipper (built in the front) (c) Zipper (built at the end)

Fig. 9: Construct a building block for a 3-pass simultaneous expandable message

The second-preimage attack on the Zipper hash. When the Zipper hash combines $k > 2$ hash functions by alternatively forward computing and reverse computing, one can still extend Attack 5.

To construct a simultaneous expandable message adaptable for the Zipper hash, one first constructs partial building blocks for all the computational passes in one direction and then constructs them for all the computational passes in the other direction. Note that in each of the building blocks in a simultaneous expandable message for the Zipper hash, there are two pieces of Joux's multi-collisions, each at one of the two ends of the building block. The left Joux's multi-collisions are for synchronizing the reverse computations, and the right Joux's multi-collisions are for synchronizing the forward computations. The complexity to build a cascade simultaneous expandable message for k hash functions can be the same as that of building a parallel one, which is approximately $O(2^{\ell} + n^{2(k-1)} \cdot 2^{n/2})$. Note that for even k, the simultaneous expandable message is placed at the end of each pass, and the length of the second preimage encoded in the simultaneous expandable message can be chosen to optimize the attack, while for odd k, the simultaneous expandable message is placed in the front end of each pass (see Fig.9b and 9c).

Next, we take $k = 3$ for example to briefly describe the extended attack (see Fig.10). The attack first collects a triple of cyclic nodes $(\bar{x}, \bar{y}, \bar{z})$, each node located in each of the three random functional graphs generated using the three compression functions. Start from $\bar{z}$, we build a $2^{r \cdot \frac{n}{2}}$-Joux's multi-collisions $\mathcal{M}_{\texttt{MC3}}$, ending with $\hat{z}$. By exploiting messages in $\mathcal{M}_{\texttt{MC3}}$, we can start from $\bar{x}$ and build a 2^r-Joux's multi-collisions $\mathcal{M}_{\texttt{MC1}}$, ending with $\hat{x}$. This is essentially computing a 2^r-simultaneous Joux's multi-collision starting from state pair $(\bar{x}, \bar{z})$ and ending with $(\hat{x}, \hat{z})$. We denoted it by $\mathcal{M}_{\texttt{MC13}}$. Start from $\bar{y}$, we build a 2^r-Joux's multi-collisions $\mathcal{M}_{\texttt{MC2}}$ for the reverse computation, ending with $\hat{y}$. Then, find a message block $\bar{m}$ mapping $\hat{z}$ to an internal state c_p in the original computation with $\boldsymbol{M}$. Afterwards, the suffix of the second preimage is fixed to be $m_{p+1} \parallel m_{p+2} \parallel \cdots \parallel m_L$.

Starting from $\hat{y}$, we compute the cascade simultaneous expandable message. Note that, only after we completed the computation in the second pass (which is a reverse computation in the middle of the two forward computations) and get the terminal point $\ddot{y}$, can we start the computation in the first and third passes. The first pass starts from IV and the third pass starts from the terminal point of the second pass, *i.e.*, $\ddot{z} = \ddot{y}$. Because there is only one reverse pass, in each building block, only the right piece of Joux's multi-collision is required (see Fig.9b, if $k \geq 4$, there should be another piece of Joux's multi-collision at the left end in each building block). At last, we launch a meet-in-the-middle procedure using messages in $\mathcal{M}_{\texttt{MC13}}$ and $\mathcal{M}_{\texttt{MC2}}$, to find a triple of starting nodes $(\check{x}, \check{y}, \check{z})$ simultaneously reach the triple of deep-iterates $(\bar{x}, \bar{y}, \bar{z})$, and output the concatenation of the obtained message fragments.

We analysis the complexity of the extended attack in general supposing that there are k computational passes. Note that, among the k computation passes, there are $\lceil k/2 \rceil$ forward computations and $\lfloor k/2 \rfloor$ reverse computations. We have

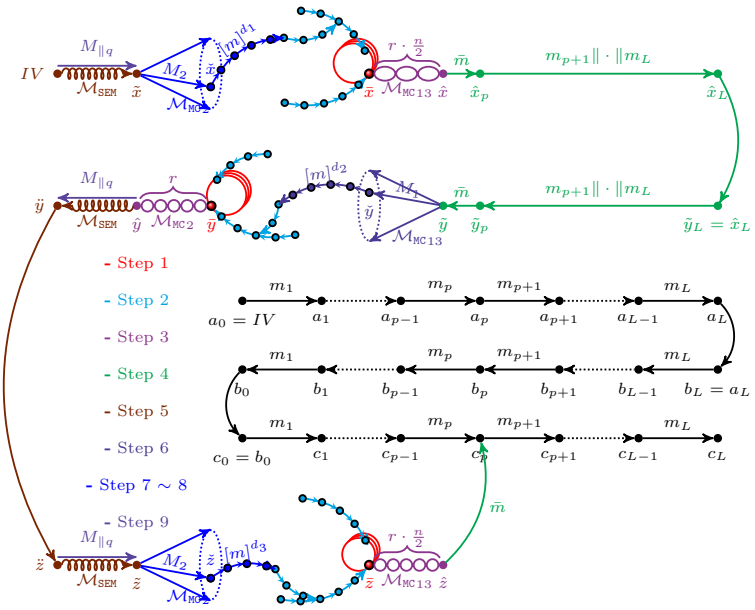


Fig. 10: Second-preimage attacks on 3-pass Zipper

2^{2r} pairs of starting nodes in the meet-in-the-middle procedure in the extended attack, which is the Cartesian product between the set of 2^r nodes for the forward computations and the set of 2^r nodes for the reverse computations. Let the deep-iterates nodes be of depth 2^d; then, each pair succeeds with probability $2^{(k+1)d-kn}$ without using multi-cycles, and the probability amplified to be $2^{(k+1)n/2-kn-n/2+\ell'}$ by using multi-cycles.

If the message length is limited to be no more than $2^{n/2}$, we cannot use the multi-cycles technique. The success of the attack requires $2r = kn - (k+1) \cdot d$, and the attack complexity is approximately $2^t + 2^{\ell'} + 2^{n-\ell} + 2^r \cdot 2^{n-t}$. Setting $d = n/2$ and $\ell' = \ell = n/2$, we get $r = kn/4 - n/4$, and the optimal complexity is $2^{kn/8+3n/8}$, which is less than 2^n for $k < 5$. Concretely, the complexity is $2^{3n/4}$ for $k = 3$ and $2^{7n/8}$ for $k = 4$.

If the message length is not limited, we can use the multi-cycles technique. The success of the attack requires $2r = kn-(k+1)\cdot n/2+n/2-\ell'$, *i.e.*, $r = kn/4-\ell'/2$. The attack complexity is approximately $2^t + 2^{\ell'} + 2^{n-\ell} + 2^{kn/4-\ell'/2+n-t}$. We balance the terms by setting $t = kn/4 - \ell'/2 + n - t = \ell'$ and achieve the optimized complexity $2^{kn/10+2n/5}$ for $n - \ell < \ell'$ and $\ell' = kn/10 + 2n/5$. It is less than 2^n when $k < 6$. Concretely, the complexity is $2^{7n/10}$, $2^{4n/5}$, $2^{9n/10}$ for $k = 3,\ 4,\ 5$ respectively.

The second-preimage attack on Hash-Twice. Attack 6 can be extended using two different strategies: either we only build an interchange structure for two of the k computational passes or we build an interchange structure to control all the k computational passes. The second strategies cannot be more efficient

than the first. Thus, we only describe the extended attack following the first strategy here.

In the extended attack on hashing k times, the frameworks for constructing the first and last passes are almost identical to that of the original attack (see Fig.11). We collect 2^s special nodes in the functional graph of the first hash function. We collect one special node in each of the last $k-1$ hash functions. If the message length is not limited, we select cyclic node as special nodes to be targeted (when using multi-cycles technique, we only consider the correctable distances between two of the k cycles); if the message length is limited, we select deep-iterate nodes as the special nodes to be targeted. We thus get 2^s tuples of k target nodes. From those target nodes in the last $k-1$ passes, we build a $2^{s\cdot n/2}$-simultaneous Joux's multi-collision $\mathcal{M}_{\mathtt{MC}}$ (that is, a set of messages that is Joux's multi-collision for the $k-1$ independent hash functions simultaneously, and is a $2^{s\cdot(n/2)^{k-i+1}}$-Joux's multi-collision for the i-th hash). We herd the 2^s target nodes in the first pass to a single state $\tilde{x}$ by building a diamond using messages in $\mathcal{M}_{\mathtt{MC}}$. We try to hit an internal state in the last computational pass from the endpoint of the last Joux's multi-collision. After that, suffix of the second preimage is fixed. We can then compute the final states of the first $k-1$ passes, which are also initial states of the last $k-1$ passes. We then start from the initial states in the k computational passes, compute a k-pass simultaneous expandable message (see Fig.9a). Starting from the terminal states of the simultaneous expandable message, we build an interchange structure in which two of the k passes have 2^r chains and the remaining $k-2$ passes have a single chain (which essentially is a simultaneous Joux's multi-collision for the remaining $k-2$ passes). We finally try to use the endpoints of the interchange structure to find a tuple of k starting nodes reaching one of the 2^s tuples of k target nodes at the same distance.

If the message length is limited to be no more than $2^{n/2}$, we use deep-iterates nodes (with depth 2^{n-g}, where $g \geq \max(n/2, n-\ell)$) as targeted nodes in the above extended attack. Following an analysis similar to the one in Sect. 8.3, one finds that the complexity is as follows (ignore the constant and polynomial factors):

Step 1: $k \cdot 2^g$ **Step 2:** $k \cdot 2^t$ **Step 3:** $n\sqrt{s} \cdot 2^{n/2+s/2} + s \cdot n^{k-1} \cdot 2^{n/2}$

Step 4: $2^{n-\ell} + 2^{\ell}$ **Step 5:** $2^{\ell} + n^{2k+2} \cdot 2^{n/2}$ **Step 6:** $n^{k-1} \cdot 2^{n/2+2r}$

Step 7: $2^{(k+1)g-t-s-r} + 2^{(k+1)g+t-s-r+\ell-2n} + 2^{kg+\ell-n-r}$

In Sect. 8.3, for the case $k = 2$, we balance the complexity using the first term in the formula of Step 7, because the last two terms in the formula of Step 7 is less than the first term under the best configuration. Here, for the case $k \geq 3$, we balance the complexity using the third term in the formula of Step 7, because the third term is greater than the first two terms. Specifically, we set $t = n/2 + s/2 = n/2 + 2r = kg + \ell - n - r$, *i.e.*, $t = 2kg/3 + 2\ell/3 - n/2$, $s = 4kg/3 + 4\ell/3 - 2n$, and $r = kg/3 + \ell/3 - n/2$. The complexity of the dominant terms is $2^g + 2^{2kg/3+2\ell/3-n/2} + 2^{-4kg/3+g-7\ell/3+3n}$, in which the mid term is always greater than 2^n for $k \geq 4$ under the restrictions $g \geq \max(n/2, n-\ell)$ and $\ell \leq n/2$. Thus, the attack can be more efficient than 2^n only for $k < 4$. For $k = 3$, we set

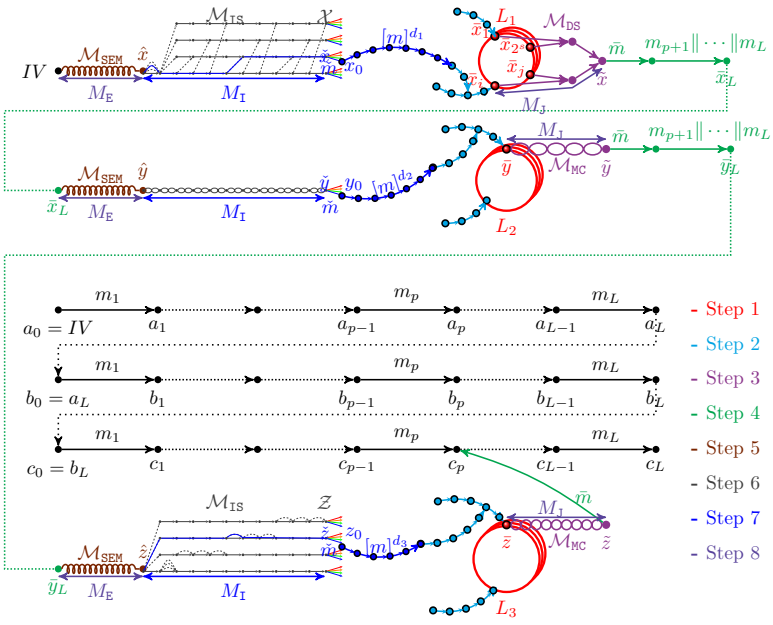


Fig. 11: Second-preimage attacks on 3-pass Hash-Twice

$g = n - \ell$ to optimize the complexity. In this case, $r = k(n-\ell)/3 + \ell/3 - n/2$, which fulfils the restriction $2r \le \ell$ when $k \le 3$. The complexity is then

$$2^{n-\ell} + 2^{(4k-3)n/6 - 2(k-1)\ell/3}.$$

It is optimally $(2k-1)n/6$, obtained for $\ell = n/2$. As a result, for $k = 3$, the optimal complexity is $2^{5n/6}$.

If the message length is not limited and $\ell > n/2$, we use the multi-cycles technique. Following an analysis similar to the one in Sect. 8.2, one can find that the complexity is as follows (ignoring the constant and polynomial factors):

Step 1: $k \cdot 2^{n/2}$ **Step 2:** $k \cdot 2^t$

Step 3: $n\sqrt{s} \cdot 2^{n/2+s/2} + s \cdot n^{k-1} \cdot 2^{n/2}$

Step 4: $2^{n-\ell} + 2^{\ell}$ **Step 5:** $2^{\ell} + n^{2k+2} \cdot 2^{n/2}$

Step 6: $n^{k-1} \cdot 2^{n/2+2r}$ **Step 7:** $2^{(k+2)n/2-t-r-s-\ell}$ **Step 8:** 2^{ℓ}

We make a balance by setting $t = n/2+s/2 = n/2+2r = (k+2)n/2-t-r-s-\ell$, *i.e.*, $t = (2k+9)n/18 - 2\ell/9$, $s = 2kn/9 - 4\ell/9$, and $r = kn/18 - \ell/9$. The complexity is then

$$2^{\ell} + 2^{(2k+9)n/18 - 2\ell/9}.$$

It is optimally $2^{(2k+9)n/22}$ when $\ell = (9+2k)n/22$. Thus, the attack costs less than 2^n for $k < 7$. Concretely, it is $2^{15n/22}$, $2^{17n/22}$, $2^{19n/22}$, $2^{21n/22}$ for $k = 3,\ 4,\ 5,\ 6$ respectively.

Remark. From the above results, in the functional-graph-based attacks, when use a set of deep iterates as targeted states for attacks with short messages, one has to use the distinguished points method, which is not adequately efficient to attack combiners of more than three hash functions. In contrast, when using a set of cyclic nodes for attacks with long messages, one does not need the distinguished points method and thus can extend the attacks to combiners with n-bit output of up to six Merkle-Damgård hash functions.

10 Summary and Open Problems

In this paper, we study the security of various of hash combiners by devising generic attacks. These attacks show rather surprising results — the security upper bounds of most hash combiners are not as high as commonly believed. Regarding basic security requirements (preimage resistance, second-preimage resistance), they fail to provide more (or even the same) security than that provided by a single ideal hash function, or even less than that provided by its underlying hash functions. See Tab. 1 for a summary of their current security status. In Fig. 12 and Fig. 13, we summarize their detailed security status by drawing trade-off curves between the length of the message and the complexity of the attacks. From these trade-off curves, for combiners with underlying hash functions using Merkle-Damgård construction, the gaps between the security upper bounds and the security lower bounds provided by security proof are quite narrow. However, that is true only for very long messages. For short messages, the gap remains large. That mainly results from the limitation of the key techniques used in our attacks. Our attacks highly exploit the iterated property of the underlying hash functions. Particularly, most of our attacks exploit properties of functional graphs of random mappings generated by fixing a message block input to the compression functions. Thus, they usually involve iterating the compression functions with fixed message block many times. Therefore, our crafted messages are very long, and they are typically composed of a large number of repeated message blocks (which can be easily recognized). Thus, one open problem is how to extend the attacks to apply to short messages or with small patches. Another open problem is how to extend the attacks to combiners with at least one underlying hash function following the HAIFA framework.

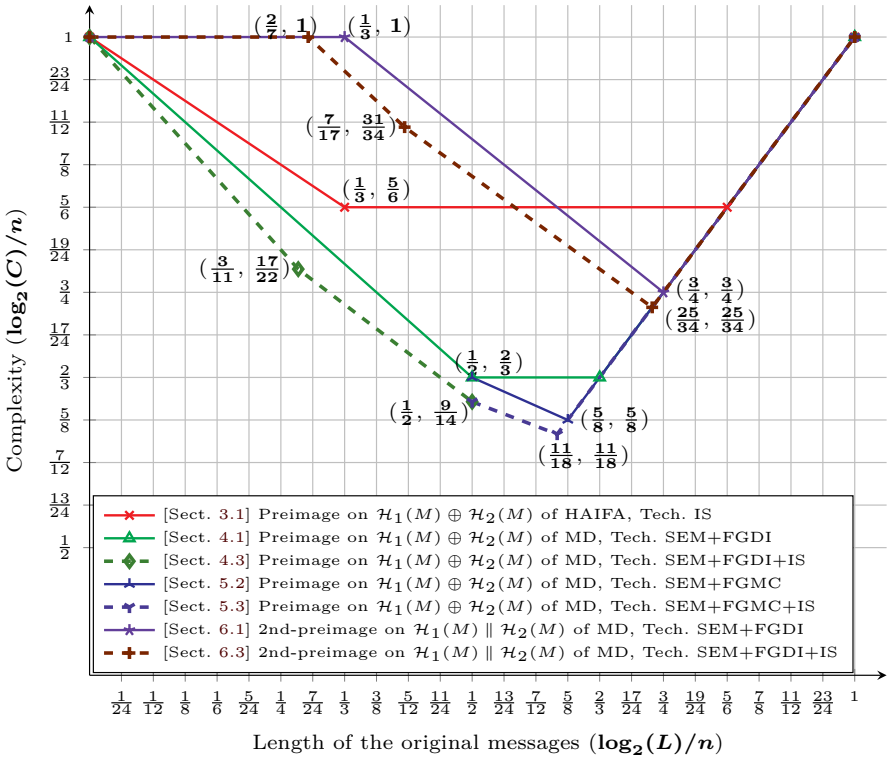


Fig. 12: Trade-offs between the message length and the complexity of attacks on parallel hash combiners

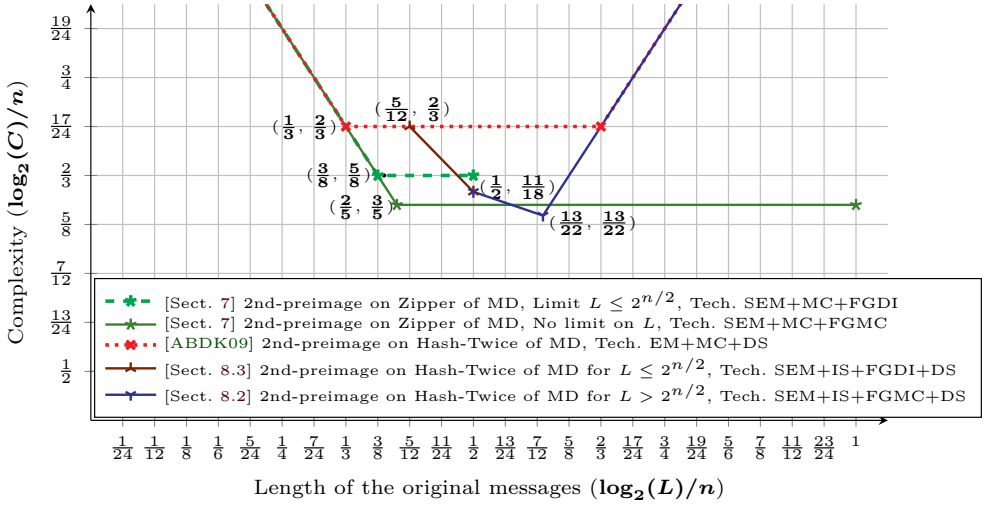


Fig. 13: Trade-offs between the message length and the complexity of attacks on cascade hash combiners

Acknowledgments

This research is supported by the National Research Foundation, Prime Minister's Office, Singapore under its Strategic Capability Research Centres Funding

Initiative, Nanyang Technological University under research grant M4082123, and Singapore's Ministry of Education under grant M4012049. Itai Dinur is supported in part by the Israeli Science Foundation through grant No. 573/16. Lei Wang is supported by National Natural Science Foundation of China (61602302, 61472250, 61672347), Natural Science Foundation of Shanghai (16ZR1416400), Shanghai Excellent Academic Leader Funds (16XD1401300), 13th five-year National Development Fund of Cryptography (MMJJ20170114).

References

ABD+16. Elena Andreeva, Charles Bouillaguet, Orr Dunkelman, Pierre-Alain Fouque, Jonathan J. Hoch, John Kelsey, Adi Shamir, and Sébastien Zimmer. New Second-Preimage Attacks on Hash Functions. *J. Cryptology*, 29(4):657–696, 2016.

ABDK09. Elena Andreeva, Charles Bouillaguet, Orr Dunkelman, and John Kelsey. Herding, second preimage and trojan message attacks beyond merkle-damgård. In Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography, 16th Annual International Workshop, SAC 2009, Calgary, Alberta, Canada, August 13-14, 2009, Revised Selected Papers*, volume 5867 of *Lecture Notes in Computer Science*, pages 393–414. Springer, 2009.

ABF+08. Elena Andreeva, Charles Bouillaguet, Pierre-Alain Fouque, Jonathan J. Hoch, John Kelsey, Adi Shamir, and Sébastien Zimmer. Second Preimage Attacks on Dithered Hash Functions. In Nigel P. Smart, editor, *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, volume 4965 of *Lecture Notes in Computer Science*, pages 270–288. Springer, 2008.

ADG+08. Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors. *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, volume 5126 of *Lecture Notes in Computer Science*. Springer, 2008.

BB06. Dan Boneh and Xavier Boyen. On the Impossibility of Efficiently Combining Collision Resistant Hash Functions. In Cynthia Dwork, editor, *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*, volume 4117 of *Lecture Notes in Computer Science*, pages 570–583. Springer, 2006.

BD07. Eli Biham and Orr Dunkelman. A Framework for Iterative Hash Functions - HAIFA. *IACR Cryptology ePrint Archive*, 2007:278, 2007.

BGW18. Zhenzhen Bao, Jian Guo, and Lei Wang. Functional Graphs and Their Applications in Generic Attacks on Iterated Hash Constructions. *IACR Trans. Symmetric Cryptol.*, 2018(1):201–253, 2018.

Bra90. Gilles Brassard, editor. *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*. Springer, 1990.

BSU12. Simon R. Blackburn, Douglas R. Stinson, and Jalaj Upadhyay. On the complexity of the herding attack and some related attacks on hash functions. *Des. Codes Cryptography*, 64(1-2):171–193, 2012.

BWGG17. Zhenzhen Bao, Lei Wang, Jian Guo, and Dawu Gu. Functional graph revisited: Updates on (second) preimage attacks on hash combiners. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part II*, volume 10402 of *Lecture Notes in Computer Science*, pages 404–427. Springer, 2017.

CJ15. Shiwei Chen and Chenhui Jin. A second preimage attack on zipper hash. *Security and Communication Networks*, 8(16):2860–2866, 2015.

Cra05. Ronald Cramer, editor. *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*. Springer, 2005.

CRS+07. Ran Canetti, Ronald L. Rivest, Madhu Sudan, Luca Trevisan, Salil P. Vadhan, and Hoeteck Wee. Amplifying Collision Resistance: A Complexity-Theoretic Treatment. In Menezes [Men07], pages 264–283.

DA99a. Richard Drews Dean and Andrew Appel. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University Princeton, 1999.

DA99b. Tim Dierks and Christopher Allen. The TLS Protocol Version 1.0. *RFC*, 2246:1–80, 1999.

Dam89. Ivan Damgård. A Design Principle for Hash Functions. In Brassard [Bra90], pages 416–427.

Din16. Itai Dinur. New Attacks on the Concatenation and XOR Hash Combiners. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 484–508. Springer, 2016.

DL14. Itai Dinur and Gaëtan Leurent. Improved Generic Attacks against Hash-Based MACs and HAIFA. In Garay and Gennaro [GG14], pages 149–168.

DP07. Orr Dunkelman and Bart Preneel. Generalizing the Herding Attack to Concatenated Hashing Schemes. In *In ECRYPT Hash Function Workshop*. Citeseer, 2007.

DR08. Tim Dierks and Eric Rescorla. The transport layer security (TLS) protocol version 1.2. *RFC*, 5246:1–104, 2008.

FKK11. Alan O. Freier, Philip Karlton, and Paul C. Kocher. The secure sockets layer (SSL) protocol version 3.0. *RFC*, 6101:1–67, 2011.

FL07. Marc Fischlin and Anja Lehmann. Security-Amplifying Combiners for Collision-Resistant Hash Functions. In Menezes [Men07], pages 224–243.

FL08. Marc Fischlin and Anja Lehmann. Multi-property Preserving Combiners for Hash Functions. In Ran Canetti, editor, *Theory of Cryptography, Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008.*, volume 4948 of *Lecture Notes in Computer Science*, pages 375–392. Springer, 2008.

FLP08. Marc Fischlin, Anja Lehmann, and Krzysztof Pietrzak. Robust Multi-property Combiners for Hash Functions Revisited. In Aceto et al. [ADG+08], pages 655–666.

FLP14. Marc Fischlin, Anja Lehmann, and Krzysztof Pietrzak. Robust Multi-Property Combiners for Hash Functions. *J. Cryptology*, 27(3):397–428, 2014.

FO89. Philippe Flajolet and Andrew M. Odlyzko. Random Mapping Statistics. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *Advances in Cryptology - EUROCRYPT '89, Workshop on the Theory and Application of of Cryptographic Techniques, Houthalen, Belgium, April 10-13, 1989, Proceedings*, volume 434 of *Lecture Notes in Computer Science*, pages 329–354. Springer, 1989.

GG14. Juan A. Garay and Rosario Gennaro, editors. *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*. Springer, 2014.

GPSW14. Jian Guo, Thomas Peyrin, Yu Sasaki, and Lei Wang. Updates on Generic Attacks against HMAC and NMAC. In Garay and Gennaro [GG14], pages 131–148.

Hel80. Martin E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Trans. Information Theory*, 26(4):401–406, 1980.

Her05. Amir Herzberg. On Tolerant Cryptographic Constructions. In Alfred Menezes, editor, *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, volume 3376 of *Lecture Notes in Computer Science*, pages 172–190. Springer, 2005.

Her09. Amir Herzberg. Folklore, practice and theory of robust combiners. *Journal of Computer Security*, 17(2):159–189, 2009.

HS06. Jonathan J. Hoch and Adi Shamir. Breaking the ICE - finding multicollisions in iterated concatenated and expanded (ICE) hash functions. In Matthew J. B. Robshaw, editor, *Fast Software Encryption, 13th International Workshop, FSE 2006, Graz, Austria, March 15-17, 2006, Revised Selected Papers*, volume 4047 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2006.

HS08. Jonathan J. Hoch and Adi Shamir. On the Strength of the Concatenated Hash Combiner When All the Hash Functions Are Weak. In Aceto et al. [ADG+08], pages 616–630.

JN15. Ashwin Jha and Mridul Nandi. Some Cryptanalytic Results on Zipper Hash and Concatenated Hash. *IACR Cryptology ePrint Archive*, 2015:973, 2015.

Jou04. Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In Matthew K. Franklin, editor, *Advances in Cryptology - CRYPTO 2004, 24th Annual International CryptologyConference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.

Jou09. Antoine Joux. *Algorithmic cryptanalysis*. Chapman and Hall/CRC, 2009.

KK06. John Kelsey and Tadayoshi Kohno. Herding Hash Functions and the Nostradamus Attack. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2006.

KS05. John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In Cramer [Cra05], pages 474–490.

Leh10. Anja Lehmann. *On the security of hash function combiners*. PhD thesis, Darmstadt University of Technology, 2010.

Lis06. Moses Liskov. Constructing an Ideal Hash Function from Weak Ideal Compression Functions. In Eli Biham and Amr M. Youssef, editors, *Selected Areas in Cryptography, 13th International Workshop, SAC 2006, Montreal, Canada, August 17-18, 2006 Revised Selected Papers*, volume 4356 of *Lecture Notes in Computer Science*, pages 358–375. Springer, 2006.

LPW13. Gaëtan Leurent, Thomas Peyrin, and Lei Wang. New Generic Attacks against Hash-Based MACs. In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II*, volume 8270 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2013.

LW15. Gaëtan Leurent and Lei Wang. The Sum Can Be Weaker Than Each Part. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 345–367. Springer, 2015.

Men07. Alfred Menezes, editor. *Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings*, volume 4622 of *Lecture Notes in Computer Science*. Springer, 2007.

Mer89. Ralph C. Merkle. One Way Hash Functions and DES. In Brassard [Bra90], pages 428–446.

Mit12. Arno Mittelbach. Hash Combiners for Second Pre-image Resistance, Target Collision Resistance and Pre-image Resistance Have Long Output. In Ivan Visconti and Roberto De Prisco, editors, *Security and Cryptography for Networks - 8th International Conference, SCN 2012, Amalfi, Italy, September 5-7, 2012. Proceedings*, volume 7485 of *Lecture Notes in Computer Science*, pages 522–539. Springer, 2012.

Mit13. Arno Mittelbach. Cryptophia's Short Combiner for Collision-Resistant Hash Functions. In Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, volume 7954 of *Lecture Notes in Computer Science*, pages 136–153. Springer, 2013.

MP14. Bart Mennink and Bart Preneel. Breaking and Fixing Cryptophia's Short Combiner. In Dimitris Gritzalis, Aggelos Kiayias, and Ioannis G. Askoxylakis, editors, *Cryptology and Network Security - 13th International Conference, CANS 2014, Heraklion, Crete, Greece, October 22-24, 2014. Proceedings*, volume 8813 of *Lecture Notes in Computer Science*, pages 50–63. Springer, 2014.

MRS09. Florian Mendel, Christian Rechberger, and Martin Schläffer. MD5 Is Weaker Than Weak: Attacks on Concatenated Combiners. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information*

Security, Tokyo, Japan, December 6-10, 2009. Proceedings, volume 5912 of *Lecture Notes in Computer Science*, pages 144–161. Springer, 2009.

NS07. Mridul Nandi and Douglas R. Stinson. Multicollision Attacks on Some Generalized Sequential Hash Functions. *IEEE Trans. Information Theory*, 53(2):759–767, 2007.

Pie07. Krzysztof Pietrzak. Non-trivial Black-Box Combiners for Collision-Resistant Hash-Functions Don't Exist. In Moni Naor, editor, *Advances in Cryptology - EUROCRYPT 2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007, Proceedings*, volume 4515 of *Lecture Notes in Computer Science*, pages 23–33. Springer, 2007.

Pie08. Krzysztof Pietrzak. Compression from Collisions, or Why CRHF Combiners Have a Long Output. In David A. Wagner, editor, *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, volume 5157 of *Lecture Notes in Computer Science*, pages 413–432. Springer, 2008.

PK14. Léo Perrin and Dmitry Khovratovich. Collision Spectrum, Entropy Loss, T-Sponges, and Cryptanalysis of GLUON-64. In Carlos Cid and Christian Rechberger, editors, *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*, volume 8540 of *Lecture Notes in Computer Science*, pages 82–103. Springer, 2014.

Pre93. Bart Preneel. *Analysis and design of cryptographic hash functions.* PhD thesis, Katholieke Universiteit te Leuven, 1993.

PW14. Thomas Peyrin and Lei Wang. Generic Universal Forgery Attack on Iterative Hash-Based MACs. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 147–164. Springer, 2014.

Rja09. Michal Rjasko. On Existence of Robust Combiners for Cryptographic Hash Functions. In Peter Vojtás, editor, *Proceedings of the Conference on Theory and Practice of Information Technologies, ITAT 2009, Horský hotel Kralova studna, Slovakia, September 25-29, 2009*, volume 584 of *CEUR Workshop Proceedings*, pages 71–76. CEUR-WS.org, 2009.

vOW99. Paul C. van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *J. Cryptology*, 12(1):1–28, 1999.

Wag02. David A. Wagner. A Generalized Birthday Problem. In Moti Yung, editor, *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 2002.

WY05. Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In Cramer [Cra05], pages 19–35.

WYY05. Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In Victor Shoup, editor, *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

A Pseudo-codes of Algorithms

Algorithm 1 Building a 2^t-Joux's Multi-Collision

function JouxMultiCollision(h, x_0, t)
  $\mathcal{M}_{\mathtt{MC}} \leftarrow \{\}$
  for $1 \leq i \leq t$ **do**
    $(x_i, m, m') \leftarrow$ Collision1(h, x_{i-1})
    $\mathcal{M}_{\mathtt{MC}} \leftarrow \mathcal{M}_{\mathtt{MC}} \parallel (m, m')$
  end for
  return $(x_t, \mathcal{M}_{\mathtt{MC}})$
end function

function Collision1(h, x)
  $\mathcal{T} \leftarrow \{\}$
  loop
    $m \leftarrow \$,\ y \leftarrow h(x, m)$
    if $\mathcal{T}[y]$ exists **then return** $(y, m, \mathcal{T}[y])$
    else $\mathcal{T}[y] \leftarrow m$
    end if
  end loop
end function

Algorithm 2 Building a single switch

function Switch(h_1, h_2, a, b, b')
  $x \leftarrow \varnothing,\ \mathcal{M}_{\mathtt{MC}} \leftarrow \emptyset$
  $(x, \mathcal{M}_{\mathtt{MC}}) \leftarrow$ JouxMultiCollision($h_1, a, n/2$)
  $\mathcal{T} \leftarrow \{\}$
  for each $M \in \mathcal{M}_{\mathtt{MC}}$ **do**
    $y \leftarrow h_2^*(b, M),\ \mathcal{T}[y] \leftarrow M$
  end for
  for each $M \in \mathcal{M}_{\mathtt{MC}}$ **do**
    $y \leftarrow h_2^*(b', M)$
    if $\mathcal{T}[y]$ exists **then**
      return $(\mathcal{T}[y], M)$
    end if
  end for
end function

Algorithm 3 Building and using a T-interchange structure

function INTERCHANGE(h_1, h_2, IV_1, IV_2)
  $a_0 \leftarrow IV_1$, $b_0 \leftarrow IV_2$
  for $1 \leq k < T$ **do**
    $a_k \leftarrow \$$, $b_k \leftarrow \$$
  end for
  for $1 \leq j < T$ **do**
    $(M, M') \leftarrow$ SWITCH(h_1, h_2, a_0, b_0, b_j)
    $\boldsymbol{M} \leftarrow \boldsymbol{M} \parallel M$, $\boldsymbol{M'} \leftarrow \boldsymbol{M'} \parallel M'$
    for $0 \leq k < T$ **do**
      $a_k \leftarrow h_1^*(a_k, M)$, $b_k \leftarrow h_2^*(b_k, M)$
    end for
  end for
  for $1 \leq j < T$ **do**
    for $1 \leq i < T$ **do**
      $(M, M') \leftarrow$ SWITCH(h_2, h_1, b_j, a_0, a_i)
      $\boldsymbol{M} \leftarrow \boldsymbol{M} \parallel M$, $\boldsymbol{M'} \leftarrow \boldsymbol{M'} \parallel M'$
      for $0 \leq k < T$ **do**
        $a_k \leftarrow h_1^*(a_k, M)$, $b_k \leftarrow h_2^*(b_k, M)$
      end for
    end for
  end for
  return $(\boldsymbol{M}, \boldsymbol{M'})$
end function

function SELECTMESSAGE($\boldsymbol{M}, \boldsymbol{M'}, j, k$)
  $\mu \leftarrow \boldsymbol{M}$
  if $k \neq 0$ **then**
    $\mu[k-1] \leftarrow \boldsymbol{M'}[k-1]$
  end if
  if $j \neq 0$ **then**
    $\mu[(k+1) \cdot (T-1) + j - 1] \leftarrow \boldsymbol{M'}[(k+1) \cdot (T-1) + j - 1]$
  end if
  return μ
end function

Algorithm 4 Constructing a building block for a simultaneous expandable message

function SEMBLOCK(x_0, y_0, i)
  $xp \leftarrow h_1^*(x_0, [0]^{i-C})$
  $(sp, m_1, m_1') \leftarrow$ COLLISION2(h_1, x_0, xp)
  $(x_1, \mathcal{M}_{\texttt{MC}}) \leftarrow$ JOUXMULTICOLLISION($h_1, sp, C-1$)
  $\mathcal{M}_{\texttt{short}} \leftarrow m_1 \times \mathcal{M}_{\texttt{MC}}$, $\mathcal{M}_{\texttt{long}} \leftarrow ([0]^{i-C} \parallel m_1') \times \mathcal{M}_{\texttt{MC}}$
  $\mathcal{Y}_{\texttt{short}} \leftarrow \emptyset$
  for each $ms' \in \mathcal{M}_{\texttt{short}}$ **do**
    $ys' \leftarrow h_2^*(y_0, ms')$, $\mathcal{Y}_{\texttt{short}} \leftarrow_{insert} (ys', ms')$
  end for
  for each $ml' \in \mathcal{M}_{\texttt{long}}$ **do**
    $yl' \leftarrow h_2^*(y_0, ml')$
    if $yl' \in \mathcal{Y}_{\texttt{short}}$ **then**
      $(y_1, ms, ml) \leftarrow (yl', ms', ml')$
    end if
  end for
  return (x_1, y_1, ms, ml)
end function

function COLLISION2(h, x, x')
  $\mathcal{T} \leftarrow \{\}$
  for $1 \le i \le 2^{n/2}$ **do**
    $m \leftarrow \$$, $y \leftarrow h(x, m)$, $\mathcal{T}[y] \leftarrow m$
  end for
  loop
    $m' \leftarrow \$$, $y' \leftarrow h(x', m')$
    if $\mathcal{T}[y']$ exists **then**
      return $(y', \mathcal{T}[y'], m')$
    end if
  end loop
end function

Algorithm 5 Expanding the functional graph of f (generating 2^t nodes in $\mathcal{FG}_f$)

procedure GEN(t)
  $\mathcal{G} \leftarrow \emptyset$
  while $|\mathcal{G}| < 2^t$ **do**
    $\mathcal{C} \leftarrow \emptyset$, $x \leftarrow_{\$} \{0, 1, \ldots, 2^n - 1\} \setminus \mathcal{G}$
    while true **do**
      if $x \in \mathcal{G}$ or $x \in \mathcal{C}$ **then**
        $\mathcal{G} \leftarrow_{merge} \mathcal{C}$, **go to line** 3
      else $\mathcal{C} \leftarrow_{insert} x$, $x \leftarrow f(x)$
      end if
    end while
  end while
  return $\mathcal{G}$
end procedure

B Optimized Interchange Structure

We now describe an optimized attack using only $(2^t-1)(2^t-1)$ switches rather than $2^{2t}-1$. The attack also requires multi-collision structures, as introduced by Joux [Jou04].

We replace the first 2^t-1 switches with a 2^t- Joux's multi-collision in $\mathcal{H}_1$, and we use those messages to initialize all the b_k chains in $\mathcal{H}_2$. We can also optimize the first series of switches in $\mathcal{H}_2$ in the same way: we build a 2^t-multi-collision in $\mathcal{H}_2$ starting from b_0, and we use those messages to initialize the a_j chains in $\mathcal{H}_1$. This is illustrated by Fig. 14, and the detailed attack is given in Alg. 6.

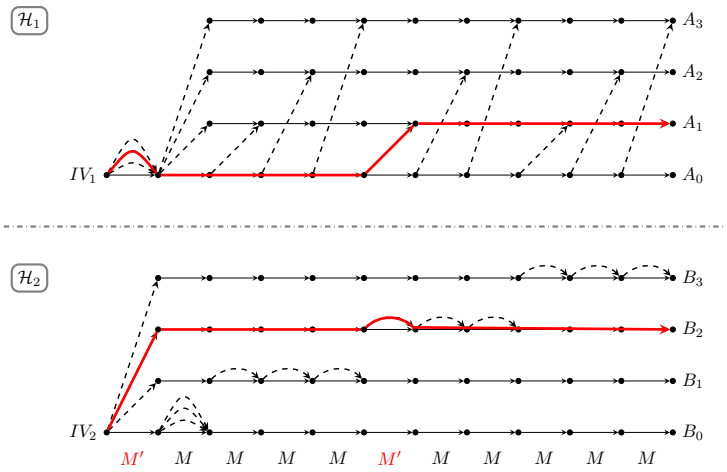


Fig. 14: Optimized interchange structure

Algorithm 6 Optimized T-interchange structure (denote $T = 2^t$)

function INTERCHANGE(h_1, h_2, IV_1, IV_2, T)
  $a_0 \leftarrow IV_1$, $b_0 \leftarrow IV_2$
  $(a_0, \mathcal{M}_{\texttt{MC0}}) \leftarrow$ JOUXMULTICOLLISION(h_1, a_0, t)
  for $0 \leq k < T$ **do**
    $b_k \leftarrow h_2^*(b_0, \mathcal{M}_{\texttt{MC0}}[k])$
  end for
  $(b_0, \mathcal{M}_{\texttt{MC1}}) \leftarrow$ JOUXMULTICOLLISION(h_2, b_0, t)
  $a_0 \leftarrow h_1^*(a_0, \mathcal{M}_{\texttt{MC1}}[0])$
  for $1 \leq k < T$ **do**
    $a_k \leftarrow h_1^*(a_0, \mathcal{M}_{\texttt{MC1}}[k])$
    $b_k \leftarrow h_2^*(b_k, \mathcal{M}_{\texttt{MC1}}[0])$
  end for
  for $2 \leq j < T$ **do**
    for $1 \leq i < T$ **do**
      $(M, M') \leftarrow$ SWITCH(h_2, h_1, b_j, a_0, a_i)
      $\boldsymbol{M} \leftarrow \boldsymbol{M} \parallel M$, $\boldsymbol{M'} \leftarrow \boldsymbol{M'} \parallel M'$
      for $0 \leq k < T$ **do**
        $a_k \leftarrow h_1^*(a_k, M)$
        $b_k \leftarrow h_2^*(b_k, M)$
      end for
    end for
  end for
  return $(\mathcal{M}_{\texttt{MC0}}, \mathcal{M}_{\texttt{MC1}}, \boldsymbol{M}, \boldsymbol{M'})$
end function

function SELECTMESSAGE($\mathcal{M}_0, \mathcal{M}_1, \boldsymbol{M}, \boldsymbol{M'}, j, k$)
  if $j = 0$ **then**
    return $\mathcal{M}_0[k] \parallel \mathcal{M}_1[0] \parallel M$
  else if $k = 0$ **then**
    return $\mathcal{M}_0[0] \parallel \mathcal{M}_1[j] \parallel M$
  else
    $\mu \leftarrow M$
    $\mu[(k-1) \cdot (T-1) + j - 1] \leftarrow M'[(k-1) \cdot (T-1) + j - 1]$
    return $\mathcal{M}_0[k] \parallel \mathcal{M}_1[0] \parallel \mu$
  end if
end function

C On Problem Raised by Dependency Between Chain Evaluations

Suppose $\bar{x}$ and $\bar{y}$ are both of depth 2^{n-g_1}. From Observation 2 in Sect. 2.7, we conclude that the probability of encountering $\bar{x}$ and $\bar{y}$ at the same distance in chains (of f_1 and f_2) evaluated from x_0 and y_0 is approximately 2^{n-3g_1}. Thus, in Sect. 4.2 and Sect. 6.2, we conclude that if the trials of chain evaluations are independent, we need to compute about 2^{3g_1-n} chains from different starting points. However, since various trials performed by selecting different starting points for the chains are dependent, it might require further proof for the conclusion.

More specifically, when the number of nodes evaluated along chains exceeding 2^{n-d}, a new chain of length 2^d is very likely to collide with a previously evaluated node due to the birthday paradox ($2^d \times 2^{n-d} = 2^n$). Thus, the outcome of this chain evaluation is determined. As a result, new chains are all related with already evaluated chains, and the dependency between them affects the outcome non-negligibly after having evaluated 2^{n-d} nodes.

However, we notice that in our attacks, the actual birthday bound for the non-negligible dependency between trials is 2^{2n-2d} instead of 2^{n-d} because in each trail, there are two chain evaluations. One is in $\mathcal{FG}_{f_1}$, and the other is in $\mathcal{FG}_{f_2}$. The chain evaluation in $\mathcal{FG}_{f_1}$ can be seen as independent with a chain evaluation in $\mathcal{FG}_{f_2}$. After having evaluated 2^{n-d} nodes in each of the two functional graphs, there is indeed a high probability for each new chain colliding with previously evaluated chains. However, for a new *pair of* chain evaluations, the probability for both chains colliding with the chains evaluated in a previous trial is significant only after having evaluated 2^{2n-2d} nodes due to the birthday paradox. That is, trials cannot be seen as independent only after having evaluated 2^{2n-2d} nodes. Note that in our attacks, required number of trials is 2^{2n-3d}, thus the total evaluated number of nodes is $2^{2n-3d} \cdot 2^{d+1} \approx 2^{2n-2d}$ which exactly falls on the birthday bound. Thus, the dependency between the trials is negligible and the complexity analysis of the corresponding attacks is justified.

From Collisions to Chosen-Prefix Collisions Application to Full SHA-1

Gaëtan Leurent[1] and Thomas Peyrin[2,3]

[1] Inria, France
[2] Nanyang Technological University, Singapore
[3] Temasek Laboratories, Singapore

gaetan.leurent@inria.fr, thomas.peyrin@ntu.edu.sg

Abstract. A chosen-prefix collision attack is a stronger variant of a collision attack, where an arbitrary pair of challenge prefixes are turned into a collision. Chosen-prefix collisions are usually significantly harder to produce than (identical-prefix) collisions, but the practical impact of such an attack is much larger. While many cryptographic constructions rely on collision-resistance for their security proofs, collision attacks are hard to turn into a break of concrete protocols, because the adversary has limited control over the colliding messages. On the other hand, chosen-prefix collisions have been shown to threaten certificates (by creating a rogue CA) and many internet protocols (TLS, SSH, IKE).
In this article, we propose new techniques to turn collision attacks into chosen-prefix collision attacks. Our strategy is composed of two phases: first a birthday search that aims at taking the random chaining variable difference (due to the chosen-prefix model) to a set of pre-defined target differences. Then, using a multi-block approach, carefully analysing the clustering effect, we map this new chaining variable difference to a colliding pair of states using techniques developed for collision attacks.
We apply those techniques to `MD5` and `SHA-1`, and obtain improved attacks. In particular, we have a chosen-prefix collision attack against `SHA-1` with complexity between $2^{66.9}$ and $2^{69.4}$ (depending on assumptions about the cost of finding near-collision blocks), while the best-known attack has complexity $2^{77.1}$. This is within a small factor of the complexity of the classical collision attack on `SHA-1` (estimated as $2^{64.7}$). This represents yet another warning that industries and users have to move away from using `SHA-1` as soon as possible.

Keywords: hash function; cryptanalysis; chosen-prefix collision; `SHA-1`; `MD5`

1 Introduction

Cryptographic hash functions are crucial components in many information security systems, used for various purposes such as building digital signature schemes, message authentication codes or password hashing functions. Informally, a cryptographic hash function H is a function that maps an arbitrarily

long message M to a fixed-length hash value of size n bits. Hash functions are classically defined as an iterative process, such as the Merkle-Damgård design strategy [Mer90, Dam90]. The message M is first divided into blocks m_i of fixed size (after appropriate padding) that will successively update an internal state (also named chaining variable), initialised with a public initial value (IV), using a so-called compression function h. The security of the hash function is closely related to the security of the compression function.

The main security property expected from such functions is *collision resistance*: it should be hard for an adversary to compute two distinct messages M and M' that map to the same hash value $H(M) = H(M')$, where "hard" means not faster than the generic birthday attack that can find a collision for any hash function with about $2^{n/2}$ computations. A stronger variant of the collision attack, the so-called *chosen-prefix* collision attack [SLdW07] is particularly important. The attacker is first challenged with two message prefixes P and P', and its goal is to compute two messages M and M' such that $H(P \parallel M) = H(P' \parallel M')$, where $\parallel$ denotes concatenation. Such collisions are much more dangerous than simple collisions in practice, as they indicate the ability of an attacker to obtain a collision even though random differences (thus potentially some meaningful information) were inserted as message prefix. In particular, this is an important threat in the key application of digital signatures: chosen-prefix collisions for `MD5` were introduced in [SLdW07], eventually leading to the creation of colliding X.509 certificates, and later of a rogue certificate authority [SSA+09]. Chosen-prefix collisions have also been shown to break important internet protocols, including TLS, IKE, and SSH [BL16], because they allow forgeries of the handshake messages.

`SHA-1` is one the most famous cryptographic hash functions in the world, having been the NIST and de-facto worldwide hash function standard for nearly two decades until very recently. Largely inspired by `MD4` [Riv91] and then `MD5` [Riv92], the American National Security Agency (NSA) first designed a 160-bit hash function `SHA-0` [Nat93] in 1993, but very slightly tweaked one part of the design two years later to create a corrected version `SHA-1` [Nat95]. It remained a standard until its deprecation by the NIST in 2011 (and disallowed to be used for digital signatures at the end of 2013). Meanwhile, hash functions with larger output sizes were standardized as `SHA-2` [Nat02] and due to impressive advances in hash function cryptanalysis in 2004, in particular against hash functions of the `MD-SHA` family [WLF+05, WY05, WYY05b, WYY05a], the NIST decided to launch a hash design competition that eventually led to the standardization in 2015 of `SHA-3` [Nat15].

There has been a lot of cryptanalysis done on `SHA-1`. After several first advances on `SHA-0` and `SHA-1` [CJ98, BCJ+05], researchers managed to find for the first time in 2004 a theoretical collision attack on the whole hash function, with an estimated cost of 2^{69} hash function calls [WYY05a]. This attack was extremely complex and involved many details, so the community worked on better evaluating and further improving the actual cost of finding a collision on `SHA-1` with these new techniques. Collisions on reduced-step versions of `SHA-1` were

computed [DMR07, KPS15], or even collisions on the whole SHA-1 compression function [SKP16], which eventually led to the announcement in 2017 of the first SHA-1 collision [SBK+17]. The best known results are listed in Table 1: the best collision attack against SHA-1 has complexity 2^{61} on CPU [Ste13], and $2^{64.7}$ when implemented on GPU [SBK+17], while the best chosen-prefix collision attack has complexity $2^{77.1}$ [SKP16].

Even though SHA-1 has been broken in 2004, it is still deployed in many security systems, because collision attacks do not seem to directly threaten most protocols, and migration is expensive. Web browsers have recently started to reject certificates with SHA-1 signatures, but there are still many users with older browsers, and many protocols and softwares that allow SHA-1 signatures. Concretely, it is still possible to buy a SHA-1 certificate from a trusted CA, and many email clients accept a SHA-1 certificate when opening a TLS connection. SHA-1 is also widely supported to authenticate TLS handshake messages.

Main SHA-1 cryptanalysis techniques. Attacks against SHA-1 are based on differential cryptanalysis, where an attacker manages to somewhat control the output of the compression function. Several important ideas were used to turn differential cryptanalysis into an effective tool against hash functions:

Linearization [CJ98]. In order to build differential trails with high probability, a linearized version of the step function is used. Differential trails with a low-weight output difference δ_O can be used to find near-collisions in the compression function (i.e. two outputs that are close to a collision, the distance metric being for example the Hamming distance).

Message modification [BC04, WYY05a] In a differential attack against a hash function, the attacker can choose messages that directly satisfy some of the constraints of the path, because there is no secret key. While the conditions in the first steps are easy to satisfy, more advanced techniques have been introduced to extend the usage of these degree of freedom to later rounds in order to speed-up collision search: neutral bits (firstly introduced for cryptanalysis SHA-0 [BC04, BCJ+05]), message modifications [WYY05a] and boomerangs/tunnels [Kli06, JP07].

Non-linear trails [WYY05a]. In order to get more flexibility on the differential trails, the first few steps can use non-linearity instead of following the constraints of the linearized step function. This does not affect the complexity of the search for conforming messages (thanks to messages modification techniques), but it allows to build trails from an arbitrary input difference to a good fixed output difference δ_O (or its opposite).

Multi-block technique [CJ98, WYY05a]. The multi-block technique takes advantage of the Davies-Meyer construction used inside the compression function. Indeed, it can be written as $h(x, m) = x + E_m(x)$ where E is a block cipher, and $+$ denotes some group operation. Because of this feed-forward, an attacker can use several differential trails in E, and several near-collisions blocks, to iteratively affect the internal state. In particular, using non-linearity in the first steps, he can derive two related trails $0 \overset{\delta_M}{\rightsquigarrow} \delta_O$ and

$\delta_O \overset{-\delta_M}{\rightsquigarrow} -\delta_O$ in E from a single linear trail, by swapping the message pair. When conforming messages are found for each block, this leads to a collision because the internal state differences cancel out (see Figure 1).

Birthday phase for chosen-prefix collisions [SLdW07, Ste13]. Differential techniques have also been used for chosen-prefix collision attacks. An attacker can relax the last steps of the differential trail to allow a set $\mathcal{D}$ of output differences rather than a single δ_O. He can also use several differential trails, and use the union of the corresponding sets. Starting from two different prefixes P, P', the chosen-prefix collision attack has two stages (see Figure 2):

- In the *birthday stage*, the attacker uses a generic collision search with message blocks m_0, m'_0 to reach a difference $\delta = H(P' \,\|\, m'_0) - H(P \,\|\, m_0)$ in $\mathcal{D}$ with complexity roughly $\sqrt{\pi \cdot 2^n / |\mathcal{D}|}$.
- In the *near-collision stage*, he builds a differential trail $\delta \rightsquigarrow -\delta$ using non-linearity in the first steps, and searches a conforming message to build the chosen-prefix collision.

Multi-block for chosen-prefix collisions [SLdW07]. If a collection of differential trails affecting separate parts of the internal state is available, chosen-prefix collision attacks can be greatly improved. In particular, if an arbitrary input difference δ_R can be decomposed as $\delta_R = -\left(\delta_O^{(1)} + \delta_O^{(2)} + \cdots + \delta_O^{(r)}\right)$, where each $\delta_O^{(i)}$ can be reached as the output of a differential trail, the attacker just has to find near-collision blocks with output differences $\delta_O^{(1)}, \ldots, \delta_O^{(r)}$ (see Figure 3).
Alternatively, if this only covers a subset of input differences, the multi-block technique is combined with a birthday stage.

Our contributions. In this work, we describe new chosen-prefix collision attacks against the `MD-SHA` family, using several improvements to reduce the complexity.

1. The main improvement comes from how we use multiple near-collision blocks. For instance, using two blocks we can start from any difference in the set $\mathcal{S} := \{\delta_1 + \delta_2 \mid \delta_1, \delta_2 \in \mathcal{D}\}$, and cancel it iteratively with a first block following a trail $\delta_1 + \delta_2 \rightsquigarrow -\delta_1$ and a second block following a trail $\delta_2 \rightsquigarrow -\delta_2$ (see Figure 4). The set $\mathcal{S}$ grows with the number of blocks: this reduces the cost of the birthday search in exchange for a more expensive near-collision stage. While there are previous chosen-prefix collision attacks using several near-collision blocks [LPRR07, Pey07, SLdW07, SSA+09, MRS15], these attacks use a collection of differential trails to impact different parts of the state (each block uses a different trail). On the opposite, our technique can be used with a single differential trail, or a collection of trails without any special property. In particular, previous chosen-prefix collision attacks based on a single trail (against `SHA-1` [Ste13] and `MD5` [SSA+09, Section 6]) used only one near-collision block.

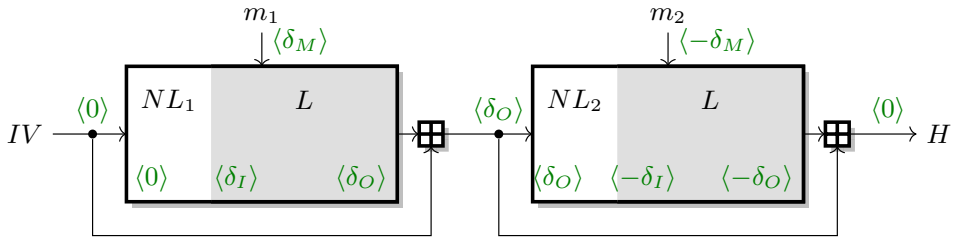


Fig. 1. 2-block collision attack using a linear trail $\delta_I \overset{\delta_M}{\rightsquigarrow} \delta_O$ and two non-linear trails $0 \rightsquigarrow \delta_I$ and $\delta_O \rightsquigarrow -\delta_I$. Green values between bracket represent differences in the state.

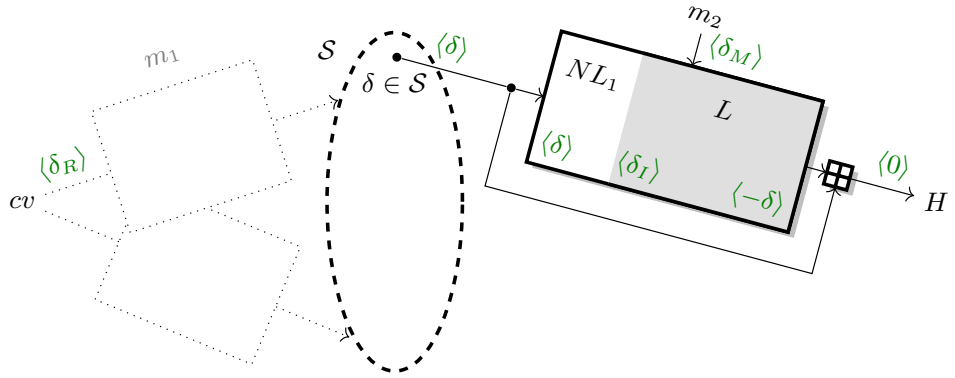


Fig. 2. Single-block chosen-prefix collision attack with a birthday stage. The linear trail $\delta_I \rightsquigarrow \delta_O$ is relaxed to reach a set $\mathcal{S}$ of feasible differences.

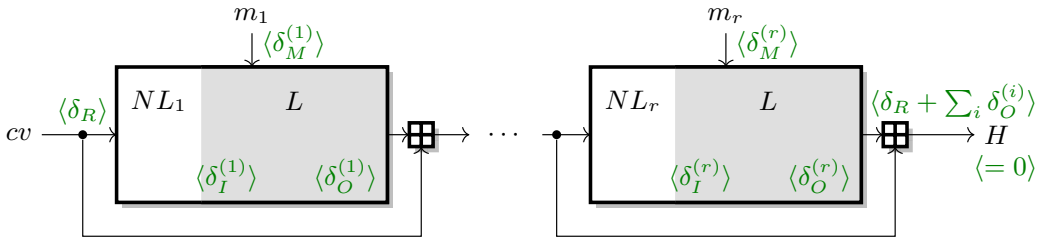


Fig. 3. Multi-block chosen-prefix collision attack. We assume that an arbitrary difference δ_R can be decomposed as $\delta_R = -\left(\delta_O^{(1)} + \delta_O^{(2)} + \cdots + \delta_O^{(r)}\right)$, where each $\delta_O^{(i)}$ can be reached as the output of a differential trail.

2. In addition, we use a clustering technique to optimize the near-collision stage, taking advantage of multiple ways to cancel a given difference. For instance, let's assume that we have to cancel a difference δ in the internal state that can be written in two different ways: $\delta = \delta_1 + \delta_2 = \delta'_1 + \delta'_2$, with $\delta_1, \delta'_1, \delta_2, \delta'_2 \in \mathcal{D}$, knowing trails $\delta \rightsquigarrow -\delta_1$ and $\delta \rightsquigarrow -\delta'_1$ with the same message constraints. Then, an attacker can target simultaneously $-\delta_1$ and $-\delta'_1$ for the first near-collision block (and use either a trail $\delta_2 \rightsquigarrow -\delta_2$ or $\delta'_2 \rightsquigarrow -\delta'_2$ for the second block, depending on the first block found). This can reduce the cost of finding the first block by a factor two.
   This technique can be seen as a generalization of an optimisation used for collision attacks with two blocks, where the first is less constrained and several output differences are allowed (for instance the `SHA-1` collision attack of [Ste13] allows 6 output differences, so that the first block is 6 times less expensive than the second).

Using these techniques, we obtain significant improvements to chosen-prefix collision attacks against `MD5` and `SHA-1`.

Application to `MD5`. We use multiple near-collision blocks to improve the complexity of the chosen-prefix collision attack with a single near-collision block given in [SSA+09, Section 6]. We start with the same differential trail, and a set $\mathcal{D}$ of size $2^{25.2}$, built in the same way. Using two near-collision blocks, we can target the set $\mathcal{S} := \{\delta_1 + \delta_2 \mid \delta_1, \delta_2 \in \mathcal{D}\}$ which contains $2^{37.1}$ elements. This leads to an attack with complexity roughly $\sqrt{\pi \cdot 2^{128}/2^{37.1}} \approx 2^{46.3}$, while the best previous attack with two blocks or less required $2^{53.2}$ MD5 computations. However, the best chosen-prefix collision attack against `MD5` is still the attack from [SSA+09] with complexity $2^{39.1}$ using 9 near-collision blocks.

Application to `SHA-1`. For `SHA-1`, we start with the attack of Stevens [Ste13], and after using several improvements we obtain a chosen-prefix collision attack with estimated complexity between $2^{66.9}$ and $2^{69.4}$ `SHA-1` computations. This is within a small factor of the complexity of a plain collision attack, estimated at $2^{64.7}$ on average [WYY05a, SBK+17], and orders of magnitude better than the $2^{77.1}$ computations cost of the currently best known chosen-prefix collision attack [Ste13] on `SHA-1`. We have conducted tests to check that our assumptions are indeed verified in practice.

First, we use a more relaxed version of the differential trail than used in [Ste13], so that we have a set $\mathcal{S}$ of size 8768 rather than 192. This directly reduce the attack cost by a factor 6.75, down to $2^{74.3}$. Next, we use the multi-block technique to build a large set $\mathcal{S}$ and to reduce further the cost of the birthday stage. Using a set $\mathcal{S}$ of size $2^{29.4}$ with a near-collision cost at most $12 \times 2^{64.7}$, this reduces the cost of the attack down to $2^{68.6}$ (with an optimistic estimate). Finally, we use the clustering technique to reduce the near-collision cost. After optimization, we have a set $\mathcal{S}$ of $2^{32.67}$ differences that can be reached with a maximum cost of $3.5 \times 2^{64.7}$ (with an optimistic estimate), leading to a full attack with complexity $2^{66.9}$ — about five time more expensive than the collision attack.

Function	Collision type	Complexity	Ref.
SHA-1	free-start collision	$2^{57.5}$	[SKP16]
	collision	2^{69}	[WYY05a]
		$2^{64.7}$	[Ste13, SBK+17][a]
	chosen-prefix collision	$2^{77.1}$	[Ste13]
		$2^{66.9}$ — $2^{69.4}$	New
MD5	collision	2^{40}	[WY05]
		2^{16}	[SSA+09]
	chosen-prefix collision (9 blocks)	$2^{39.1}$	[SSA+09]
	(3 blocks)	2^{49}	[SSA+09]
	(1 block)	$2^{53.2}$	[SSA+09]
	(2 blocks)	$2^{46.3}$	New

Table 1. Comparison of previous and new cryptanalysis results on MD5 and SHA-1. A free-start collision is a collision or the compression function only, where the attacker has full control on all the primitive's inputs. Complexities in the table are given in terms of SHA-1 equivalents on a GTX-970 GPU (when possible).

[a] The attack has a complexity of 2^{61} on CPU, and $2^{64.7}$ on GPU

Our result is surprising since we show that the cost to find a chosen-prefix collision for SHA-1 is not much more than a simple collision search. Moreover our work has a strong impact in practice as chosen-prefix collision attacks are much more dangerous than simple collisions (see for example the case of MD5 [SSA+09]). This is yet another warning that SHA-1 should be totally removed from any security product as soon as possible. The thinking "a collision attack is not directly exploitable, thus we are fine" is clearly wrong for SHA-1, and we give a proof here.

Our method is in essence quite generic, even though a lot of details have to be taken care of in order to make it work. Since most collision attacks on members of the MD-SHA family are built on the same principles as SHA-1 attacks, we believe similar ideas would apply and a collision attack can probably be transformed into a chosen-prefix collision attack for a reasonable extra cost factor. We do not foresee any reason why this technique would not apply to non MD-SHA hash functions as well (except wide-pipe hash functions which would make the birthday part too costly).

Outline. We first consider the impact of this result and give some recommendations in Section 2. Then, we describe SHA-1 and previous cryptanalysis works on this hash function in Section 3. The generic high-level description of our attack is given in Section 4, while the details regarding its application to MD5 and SHA-1 are provided in Sections 5 and 6, respectively. Eventually, we conclude and propose future works in Section 7.

2 Implications and Recommendations

Our work shows that finding a chosen-prefix collision is much easier than previously expected, and potentially not much harder than a normal collision search for SHA-1. As a real collision has already been computed for this hash function, one can now assume that chosen-prefix collisions are reachable even by medium funded organisations. Since a chosen-prefix collision attack can break many widely used protocols, we strongly urge users to migrate to SHA-2 or SHA-3, and to disable SHA-1 to avoid downgrade attacks.

Cost Estimation. We use the same estimation process as in [SBK+17]. With an optimistic spot-price scenario on g2.8xlarge instances of Amazon EC2, the authors estimated that the workload spent to find the SHA-1 collision was equivalent to a cost of about US$ 110 K, with $2^{63.4}$ SHA-1 equivalent calls on GTX-970 GPUs. We recall that they found the collision with less computations than expected. Using expected computational cost, the average workload required to find a SHA-1 collision is equivalent to a cost of about US$ 275 K, or $2^{64.7}$ SHA-1 calls. An optimistic analysis of our attack leads to a complexity of $2^{66.9}$ SHA-1 equivalent calls on GTX-970 GPUs, corresponding to a cost of US$ 1.2 M, while a more conservative analysis yields a complexity of $2^{69.4}$, or a cost of US$ 7M.

Hardware will improve as well as cryptanalysis and we can expect that collision together with chosen-prefix collision attacks will get cheaper over the years. Migration from SHA-1 to the secure SHA-2 or SHA-3 hash algorithms should now be done as soon as possible, if not already.

Impact of Chosen-prefix Collisions. Chosen-prefix collision attacks have been demonstrated already for MD5, and they are much more dangerous that identical-prefix collision attacks, with a strong impact in practice. In particular, chosen-prefix collision attack on MD5 have been used to generate colliding X.509 certificates [SLdW07] and later a rogue certificate authority [SSA+09], undermining the security of websites. They have also been shown to threaten important internet protocols, including TLS, IKE, and SSH using forgery of handshake messages. The SLOTH attacks [BL16] can break various security properties of these protocols using MD5 chosen-prefix collisions, such as client impersonation in TLS 1.2. MD5 has now been removed from most security applications, but the very same threats are now a reality for SHA-1.

However, after these attacks were demonstrated with MD5, countermeasures have been implemented that make the attacks harder to exploit. In particular, CAs are now expected to randomize the serial number of X.509 certificates, which prevents the attack against certificates. Similarly, IKEv2 implementations randomize the SPI of each connection, forcing an attacker to compute the chosen-prefix collision online, after receiving the SPI. Many implementations are also more strict with suspicious looking messages and parameters (such as the cookies used in the IKEv2 attack), and SHA-1 has been depreciated from some protocols (in particular, RFC 8247 depreciates SHA-1 in IKEv2, and SHA-1 has been removed from TLS 1.3).

As far as we understand, attacks with an online chosen-prefix collision should still be possible against the SSH and TLS 1.2 handshake. This is still far from practical[4], but it is nonetheless significantly easier than it should be with sound cryptographic primitives. Moreover, the IKEv2 and X.509 attacks might still be possible against implementations that still use SHA-1 and don't use proper countermeasures, but this requires a careful analysis of every implementation.

Usage of `SHA-1`. Even if practically broken only very recently, `SHA-1` has been theoretically broken since 2004. It is therefore surprising that `SHA-1` remains deployed in many security systems. In particular, as long as `SHA-1` is *allowed*, even if it is not used in normal operation, an attacker can use weaknesses in `SHA-1` to forge a signature, and the signature will be accepted.

First, `SHA-1` is still widely used to authenticate handshake messages in secure protocols such as TLS, SSH or IKE. As shown with the SLOTH attack [BL16], this allows various attacks using chosen-prefix collision, such as breaking authentication. These protocols have removed support for `MD5` after the SLOTH attack, but `SHA-1` is still widely supported. Actually, more than[5] 5% of the web servers from Alexa's top 1M (including `skype.com`) *prefer* to use `SHA-1` to authenticate TLS handshake messages.

An important effort is underway to remove `SHA-1` certificates from the Web, and major browsers are now refusing to connect to servers still using `SHA-1`-based certificates. Yet `SHA-1`-based certificates remains present: according to scans of the top 1 million websites from Alexa by censys.io, there are still about 35 thousand[6] servers with `SHA-1` certificates out of 1.2 million servers with HTTPS support. `SHA-1`-based certificates are also used with other protocols: for instance 700 thousand[7] out of 4.4 million mail servers (with IMAPS) use a `SHA-1` certificate. Actually, it is still possible to buy a `SHA-1` certificate from a trusted root[8]! Even though recent web browsers reject those certificates, they are accepted by older browsers and by many clients for other protocols. For instance, the "Mail" application included in Windows 10 still accepts `SHA-1` certificates without warnings when opening an IMAPS connection.

Unfortunately, many industry players did not consider moving away from `SHA-1` a priority, due to important costs and possible compatibility and bug issues induced by this move. An often-heard argument is that a simple collision attacks against a hash function is not very useful for an attacker, because he

[4] Using our attack, it would take a computation power similar to the full bitcoin network to compute a `SHA-1` chosen-prefix collision within one minute, and some parts of the attack will be hard to parallelize to such a degree.

[5] https://censys.io/domain?q=tags:https+and+443.https.tls.signature.hash_algorithm:sha1

[6] https://censys.io/domain?q=tags:https+and+443.https.tls.certificate.parsed.signature_algorithm.name:SHA1*

[7] https://censys.io/ipv4?q=tags:imaps+and+993.imaps.tls.tls.certificate.parsed.signature_algorithm.name:SHA1*

[8] https://www.secure128.com/online-security-solutions/products/ssl-certificate/symantec/sha-1-private-ssl/

doesn't have much control over the colliding messages. Therefore, there seemed to be a long way to go before really useful collision attacks would be found for SHA-1, if ever. Indeed, the current best chosen-prefix collision attack against SHA-1 requires $2^{77.1}$ hash calls [Ste13], thus orders of magnitude harder than the cost of finding a simple collision. Similarly, in the case of MD5, the cost goes from 2^{16} to 2^{39} for the currently best known collision and chosen-prefix collision attacks. However, this is a dangerous game to play as the history showed that cryptanalysis only keep improving, and attackers will eventually come up with ways to further improve their cryptanalysis techniques. For example, in the case of MD5, collisions for the compression function were found [dB94] in 1993, collisions for the whole hash function were found [WY05] in 2004, colliding X.509 MD5-based certificates were computed [SLdW07] in 2007, and rogue Certificate Authority certificate [SSA+09] was eventually created in 2009.

3 Preliminaries

3.1 Description of SHA-1

We describe here the SHA-1 hash function, but we refer to [Nat95] for all the complete details.

SHA-1 is a 160-bit hash function based on the well-known Merkle-Damgård paradigm [Dam89, Mer89]. The message input is first padded (with message length encoded) to a multiple of 512 bits, and divided into blocks m_i of 512 bits each. Then, each block is processed via the SHA-1 compression function h to update a 160-bit chaining variable cv_i that is initialised to a constant and public initial value (IV): $cv_0 = IV$. More precisely, we have $cv_{i+1} = h(cv_i, m_{i+1})$. When all blocks have been processed, the hash output is the last chaining variable.

The compression function is similar to other members of the MD-SHA family of hash functions. It is based on the Davies-Meyer construction, that turns a block cipher E into a compression function: $cv_{i+1} = E_{m_{i+1}}(cv_i) + cv_i$, where $E_k(y)$ is the encryption of the plaintext y with the key k, and $+$ is a word-wise modular addition.

The internal block cipher is composed of 80 steps (4 rounds of 20 steps each) processing a generalised Feistel network. More precisely, the state is divided into five registers $(A_i, B_i, C_i, D_i, E_i)$ of 32-bit each. At each step, an extended message word W_i updates the registers as follows:

$$\begin{cases} A_{i+1} = (A_i \lll 5) + f_i(B_i, C_i, D_i) + E_i + K_i + W_i \\ B_{i+1} = A_i \\ C_{i+1} = B_i \ggg 2 \\ D_{i+1} = C_i \\ E_{i+1} = D_i \end{cases}$$

where K_i are predetermined constants and f_i are boolean functions (in short: IF function for the first round, XOR for the second and fourth round, MAJ for the third round, see Table 2). Since only a single register value is updated

(A_{i+1}), the other registers being only rotated copies, we can express the SHA-1 step function using a single variable:

$$A_{i+1} = (A_i \lll 5) + f_i(A_{i-1}, A_{i-2} \ggg 2, A_{i-3} \ggg 2) + (A_{i-4} \ggg 2) + K_i + W_i.$$

For this reason, the differential trails figures in this article will only represent A_i, the other register values at a certain point of time can be deduced directly.

step i	$f_i(B, C, D)$	K_i
$0 \leq i < 20$	$f_{IF} = (B \wedge C) \oplus (\overline{B} \wedge D)$	0x5a827999
$20 \leq i < 40$	$f_{XOR} = B \oplus C \oplus D$	0x6ed6eba1
$40 \leq i < 60$	$f_{MAJ} = (B \wedge C) \oplus (B \wedge D) \oplus (C \wedge D)$	0x8fabbcdc
$60 \leq i < 80$	$f_{XOR} = B \oplus C \oplus D$	0xca62c1d6

Table 2. Boolean functions and constants of SHA-1

The extended message words W_i are computed linearly from the incoming 512-bit message block m, the process being called message extension. One first splits m into 16 32-bit words $M_0, \ldots, M_{15}$, and then the W_i's are computed as follows:

$$W_i = \begin{cases} M_i, & \text{for } 0 \leq i \leq 15 \\ (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) \lll 1, & \text{for } 16 \leq i \leq 79 \end{cases}$$

3.2 Previous Works

Collision attacks on SHA-1. We quickly present here without details the previous advances on SHA-1 collision search. First results on SHA-0 and SHA-1 were obtained by linearizing the compression function and constructing differential trails based on the probabilistic event that difference spreads will indeed happen linearly. These linear trails are generated with a succession of so-called local collisions (small message disturbances whose influence is corrected with other message differences inserted in the subsequent SHA-1 steps) that follows the SHA-1 message expansion. However, with this linear construction, impossibilities might appear in the first 20 steps of SHA-1 (for example due to the f_{IF} boolean function that never behaves linearly in some specific situations) and the cheapest linear trail candidates might not be the ones that start and end with the same difference (which is a property required to obtain a collision after the compression function feed-forward). Thus, since [WYY05a], collision attacks on SHA-1 are performed using two blocks containing differences. The idea is to simply pick the cheapest linear trail from roughly step 20 to 80, without paying any attention to the f_{IF} constraints or to the input/output differences. Then, the attacker will generate a non-linear differential trail for the first 20 steps in order

to connect the actual incoming input difference to the linear part difference at step 20. With two successive blocks using the same linear trail (just ensuring that the output difference of the two blocks have opposite signs), one can see in Figure 1 that a collision is obtained at the end of the second block.

Once the differential trail is set, the attacker can concentrate on finding a pair of messages that follows it for each successive block. For this, he will construct a large number of messages that follow the trail up to some predetermined step, and compute the remaining steps to test whether the output difference is the required one. The computational cost is minimized by using a simple early-abort strategy for the 16 first steps, but also more advanced amortization methods such as neutral bits [BCJ+05], boomerangs [JP07] or message modification [WYY05a] for a few more steps. Usually, the first 20 or so steps do not contribute the complexity of the attack.

Chosen-prefix collision attacks. The first concrete application of a chosen-prefix collision attack was proposed in [SLdW07] for `MD5`. This work was also the first to introduce a birthday search phase in order to find such collisions. The idea is to process random message blocks after the challenged prefixes, until the chaining variable difference δ belongs to a large predetermined set $\mathcal{S}$. Since the message blocks after each prefix are chosen independently, this can be done with birthday complexity $\sqrt{\pi \cdot 2^n/|\mathcal{S}|}$. Then, from that difference δ, the authors eventually manage to reach a collision by slowly erasing the unwanted difference bits by successfully applying some near-collision blocks. We note that the starting difference set $\mathcal{S}$ during the birthday phase must not be too small, otherwise this phase would be too costly. Moreover, the near-collisions blocks must not be too expensive either, and this will of course depend on the cryptanalysis advancements of the compression function being studied.

Finally, using also this two-phase strategy, in [Ste13] is described a chosen-prefix collision attack against the full `SHA-1`, for a cost of $2^{77.1}$ hash calls. The improvement compared to the generic 2^{80} attack is not very large, due to the difficulty for an attacker to generate enough allowable differences that can later be erased efficiently with a near-collisions block. Indeed, the compression function of `SHA-1` being much stronger than that of `MD5`, few potential candidates are found. Actually, Stevens only considers one type of near collision block, following the best differential trail used for the collision attack. By varying the signs of the message and output differences, and by letting some uncontrolled differences spread during the very last steps of the compression function, a set $\mathcal{S}$ of 192 allowable differences is obtained. However, having such a small set makes the birthday part by far the most expensive phase of the attack.

In this article, we will use essentially the same strategy: a birthday phase to reach a set $\mathcal{S}$ of allowable differences and a near-collision phase to remove the remaining differences. We improve over previous works on several points. First, we further generalise for `SHA-1` the set of possible differences that can be obtained for a cheap cost with a single message block. Secondly, we propose a multi-block strategy for `SHA-1` that allows to greatly increase the size of the set $\mathcal{S}$. Finally, we study the clustering effect that appears when using a multi-

block strategy. This can be utilised by the attacker to select dynamically the allowable differences at the output of each successive blocks, to further reduce the attack complexity. Notably, and in contrary to previous works, our set $\mathcal{S}$ is not the direct sum of independent subspaces corresponding to distinct trails. On the opposite, our applications use the same core differential trail for all the near-collision blocks. Overall, we improve substantially previous attack [Ste13] from $2^{77.1}$ `SHA-1` calls to only $2^{66.9}$. Surprisingly, our attack is very close to some sort of optimal since its cost is not much greater than that of finding a simple collision. Our attack being rather generic, we believe that this might be the case for many hash functions, which contradicts the idea that chosen-prefix collisions are much harder to obtain than simple collisions.

One can mention other parallel researches conducted on finding chosen-prefix collision attacks for various hash functions. For example, in [Pey07], the author explains how to compute collisions with random incoming differences in the internal state for the `GRINDAHL` hash function, the strategy being to slowly remove these differences thanks to the many degrees of freedom available every step. Such a divide-and-conquer technique is not applicable at all to `SHA-1` as the degrees of freedom are much fewer and only available at the beginning of the compression function. In [MRS15], inspired by the second-preimage attack against `SMASH` [LPRR07], the authors proposed a chosen-prefix collision attack on a reduced version of the `GROSTL` hash function. However, this attack strongly relies on the ability of the attacker to perform a rebound attack, which seems really hard to achieve in the case of `SHA-1`.

4 From Collision to Chosen-Prefix Collision

4.1 The Chosen-prefix Collision Attack

We assume that the hash function considered is an n-bit narrow pipe primitive, based on a Merkle-Damgård-like operating mode. In addition, we assume that the compression function is built upon a block cipher in a Davies-Meyer mode.

Preparing the attack. The attacker first builds a set $\mathcal{S}$ and a graph $\mathcal{G}$; $\mathcal{S}$ corresponds to a set of differences that can be cancelled with near-collision blocks, and $\mathcal{G}$ is used to find the sequence of blocks needed to cancel a difference in $\mathcal{S}$. We first explain how to execute the attack when $\mathcal{S}$ and $\mathcal{G}$ are given, and we will explain how to build them in Section 4.2.

The prefixes (stage 1 of Figure 4). The attacker receives the two challenge prefixes and pads them to two prefixes of the same length, to avoid differences in the final length padding. After processing the two padded prefixes starting from the IV, he reaches states cv/cv', and we denote the corresponding difference as δ_R.

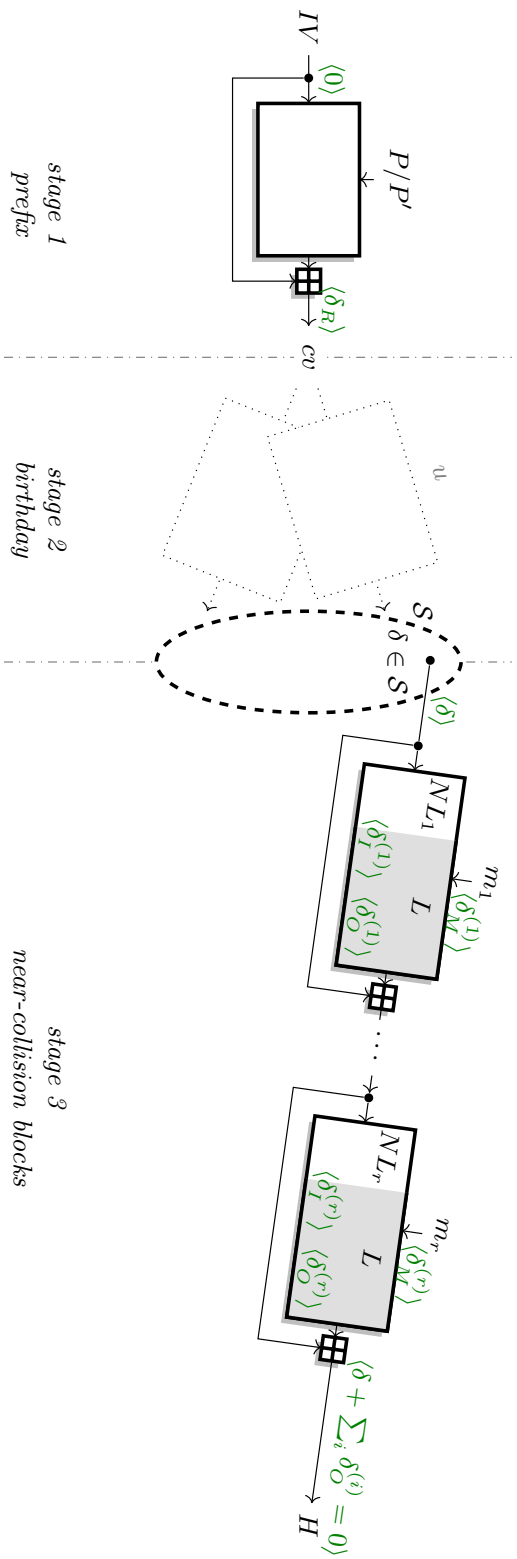


Fig. 4. High-level view of the chosen-prefix collision attack

The birthday search (stage 2 of Figure 4). The goal of the attacker is now to find one message block pair (u, u') to reach a chaining variable pair with a difference δ that belongs to $\mathcal{S}$, the set of acceptable chaining variable differences.

For this stage, we use the parallel collision search algorithm of van Oorschot and Wiener [vW99]. When a memory $M \gg C$ is available, this algorithm can find C collisions in a function $f : \{0,1\}^k \to \{0,1\}^k$ with complexity $\sqrt{\pi/2 \cdot 2^k \cdot C}$, and is efficiently parallelizable. It computes chains of iterates of the function f, and stops when the end point is a *distinguished point*, *i.e.* it satisfies some easy to detect property. In practice, we stop a chain when $x < 2^k \cdot \theta$, with $\theta \gg \sqrt{C/2^k}$, and we store on average the starting points and end points of $\theta \cdot \sqrt{\pi/2 \cdot 2^k \cdot C}$ chains (the expected length of a chain is $1/\theta$). When colliding end points are detected, we restart the corresponding chains to locate the collision point, with an expected cost of $2C/\theta$, which is small compared to the total complexity if $\theta \gg \sqrt{C/2^k}$.

In our case, we are looking for message blocks (u, u') such that $h(cv, u) - h(cv', u') \in \mathcal{S}$. Therefore, we need a function f such that a collision in f corresponds to good (u, u') with high probability. First, we consider a truncation function $\tau : \{0,1\}^n \to \{0,1\}^k$, so that pairs (x, x') with $x - x' \in \mathcal{S}$ have $\tau(x) = \tau(x')$ with high probability:

$$p = \Pr_{x,x'} \left[\tau(x) = \tau(x') \mid x - x' \in \mathcal{S}\right] \approx 1.$$

For functions of the MD-SHA family, the group operation + is a word-wise modular addition, and we build τ by removing bits that are directly affected when adding a value in $\mathcal{S}$, and bits that are affected by a carry with a relatively high probability. This typically leads to p close to one (as seen in previous attacks [SSA+09, Ste13], and the new attacks in this paper). Then, we build f as:

$$f(u) := \begin{cases} \tau(h(cv\ , \mathrm{pad}(u))) & \text{if } u[0] = 1; \\ \tau(h(cv', \mathrm{pad}(u))) & \text{else.} \end{cases}$$

The probability that a collision in f corresponds to a pair (u, u') with $h(cv, u) - h(cv', u') \in \mathcal{S}$ can be estimated as:

$$p_f = \frac{1}{2} \cdot \Pr_{x,x'} \left[x - x' \in \mathcal{S} \mid \tau(x) = \tau(x')\right] = \frac{p}{2} \cdot \frac{|\mathcal{S}|/2^n}{2^{-k}}$$

Finally, we need $1/p_f$ collisions in f, and the total complexity of the birthday stage is on average:

$$\sqrt{\frac{\pi}{2} \cdot \frac{2^k}{p_f}} = \sqrt{\frac{\pi \cdot 2^n}{p \cdot |\mathcal{S}|}} \approx \sqrt{\frac{\pi \cdot 2^n}{|\mathcal{S}|}}.$$

The multi-block collision search (stage 3 of Figure 4). The attacker now uses the graph $\mathcal{G}$ to build a sequence of near-collision blocks that ends up with a collision. Each node of the graph represents one chaining variable difference

in the set $\mathcal{S}$. To each node i of the graph is associated a cost value w_i that represents the cost an attacker will expect to pay from this particular chaining variable difference i in order to reach a colliding state (with one or multiple message blocks). Of course, a null cost will be associated with the zero difference ($w_0 = 0$). A directed edge from node i to node j represents a way for an attacker to reach chaining variable difference j from difference i with a single message block. Note that the graph is acyclic, as we will ensure that the edges will always go to strictly lower costs (i.e. an edge from i to j is only possible if $w_j < w_i$). To each edge is attached the details of the differential trail and message difference to use for that transition to happen. However, a *very* important point is that all edges going out of a node i will share the same core differential trail (by core differential trail, we mean the entire differential trail except the last steps for which one can usually accept a few divergences in the propagation of the differences). For example, during the attack, from a chaining difference i, an attacker can potentially reach difference j or difference k using the same core differential trail (and thus without having to commit in advance which of the two differences he would like to reach). Thus, in essence, the details of the differential trail and message difference to use can be directly attached to the source node.

Once the attacker hits a starting difference $\delta \in \mathcal{S}$ in the birthday phase, he will pick the corresponding node in $\mathcal{G}$, and use the differential trail and message difference attached to this node. He will use this differential trail until he reaches one of the target nodes (which has necessarily a lower expected cost attached to it). As explained in the next section, targeting several nodes simultaneously reduces the cost of the attack, because it is easier to hit one node out of many than a fixed one. We call this the clustering effect, because we use a cluster of paths in the graph. When a new node is reached, the attacker repeats this process until he eventually reaches the colliding state (i.e. null difference). Overall, the expected computational cost for this phase is the cost attached to the node δ (in practice, when actually computing one collision, he might pay a slightly lower or higher computational cost as the w_i's are expected values).

We note that any suffix message blocks that do not contain differences can of course be added after this colliding state, as the Merkle-Damgård-like mode will maintain the collision throughout the subsequent compression function calls.

4.2 Building the Set $\mathcal{S}$ and the Graph $\mathcal{G}$

We now describe how the set $\mathcal{S}$ and the graph $\mathcal{G}$ can be built during the preparation of the attack. For that, we first need to describe what an adversary can do when he tries to attack the compression function. We consider that the attacker knows some good core differential trails for the internal block cipher E, that is differential trails that go from early steps to late steps of E. For each core differential trail $\texttt{CDT}_i$ there are several possible output differences δ_j^i for E. This is typically what happens in the chosen-prefix collision attack on `SHA-1` [Ste13] where some differences in the very last steps can be allowed to spread differently than planned, thus generating new output differences. We denote the set of all possible

output differences as $\mathcal{D}$ (in particular, we have $0 \in \mathcal{D}$, and $\delta \in \mathcal{D} \iff -\delta \in \mathcal{D}$ because of symmetries).

We finally assume that any input difference for E can be mapped to any of the core differential trails inside the primitive. In the case of a `SHA-1` attack, this is achieved with the non-linear part of the trail in the first steps of the function. As shown in previous works, it allows to map any input difference to any internal difference. The non-linear part has a low probability, but during the near-collision search this is solved using the many degrees of freedom available in the first steps of the function.

Building the graph $\mathcal{G}'$. The attacker will first build a graph $\mathcal{G}'$ and filter it later to create $\mathcal{G}$. The graph $\mathcal{G}'$ is similar to the graph $\mathcal{G}$: each node will represent a chaining variable difference. A directed edge from node i to node j represents again a way for an attacker to reach chaining variable difference j from difference i with a single message block, stored with the details of the differential trail attached to it, and the cost to find the corresponding block. The differences with $\mathcal{G}$ are that (see Figure 5):

- $\mathcal{G}'$ can potentially be cyclic, as we do not ensure that an edge goes from a higher to a strictly smaller cost;
- all outgoing edges from a node i will not necessarily share the same core differential trail;
- there can be several edges from i to j, with different core differential trails.

In order to build the graph $\mathcal{G}'$, starting from the colliding state $\delta = 0$, we will simply proceed backward. We go through all possible core differential trails for E and their possible output differences δ_j^i. Due to the feed-forward of the Davies-Meyer construction, all these differences can be used to reach the colliding state by simply forcing their respective opposite $-\delta_j^i$ as input difference of the cipher (since we assumed that any input difference for E can be mapped to any of the core differential trails inside the primitive, this is always possible). Thus, for each such difference δ_j^i coming from a core differential trail $\texttt{CDT}_i$, we will add a node $-\delta_j^i$ in the graph $\mathcal{G}'$, and an edge going from this new node to the colliding state. If a node with that difference already exists in the graph, we add the edge between this node and the colliding state. This means that nodes can have several incoming and outgoing edges.

We iteratively repeat this process again with all the newly created nodes as starting points (instead of the collision state). This will create a bigger and bigger graph as we keep iterating, and the attacker can simply stop when he believes that he has enough nodes in the graph (alternatively, he can set an upper limit on the cost of the nodes to consider, or on the depth of the search, which will naturally limit the size of the graph).

The clustering effect. A simple way to build a graph $\mathcal{G}$ for the attack is to keep only the minimal cost paths in the graph $\mathcal{G}'$ (the corresponding edges form a tree), and to set the cost of the nodes to cost of the minimal path. However,

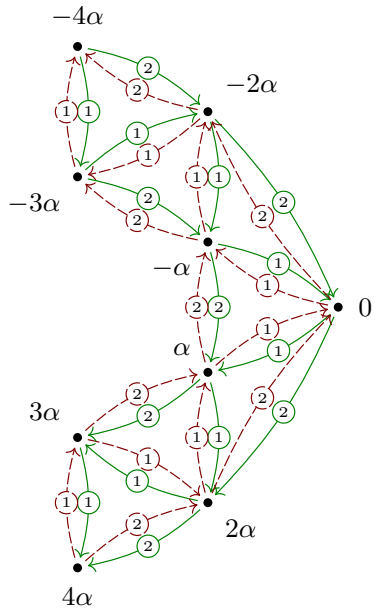


Fig. 5. Example of a graph $\mathcal{G}'$, with a bundle $\{\alpha, 2\alpha\}$ with costs 1 and 2 (solid green lines), and a bundle $\{-\alpha, -2\alpha\}$ with costs 1 and 2 (dashed red lines). The corresponding set $\mathcal{S}$ is $\{-4\alpha, -3\alpha, -2\alpha, -\alpha, 0, \alpha, 2\alpha, 3\alpha, 4\alpha\}$.

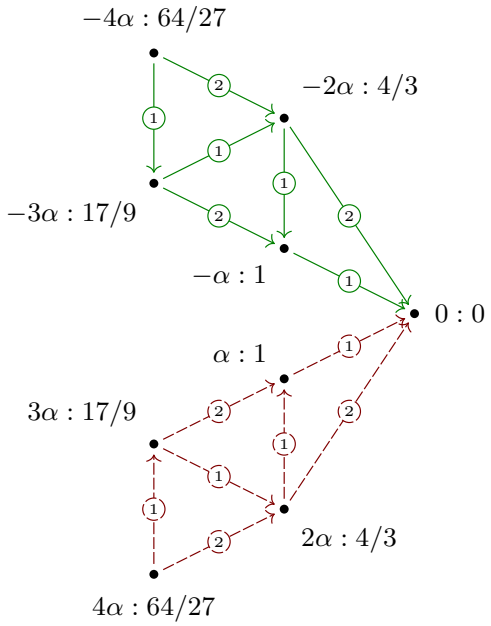


Fig. 6. The graph $\mathcal{G}$ corresponding to $\mathcal{G}'$. We show the cost of each edge and each node. In particular, note that use of clustering reduces the cost of node 4α from 4 to $64/27 \approx 2.37$

the attack cost can be greatly improved with the *clustering effect*: during the last phase of the attack, when the attacker is currently located at a node N, he does not necessarily need to choose in advance which outgoing edge of N he will try to follow. Indeed, the only commitment he needs to make at this point is which core differential trail he will use to go to the next node. Thus, he can simultaneously target several output differences corresponding to the same core differential trail, and the cost to reach one difference out of many is significantly lower than the cost of reaching any given output difference. For instance, when computing the first block of a `SHA-1` collision, Stevens [Ste13] allows six output differences with a similar cost, so that the cost to reach one of them is one sixth of the cost to reach a predetermined one.

For a given node, we call *bundle* of a core differential trail $\texttt{CDT}_i$ the grouping of all outgoing edges of that node that use $\texttt{CDT}_i$ (see Figure 5). Let $\mathcal{B}_N$ stand for the set of all bundles of a node N, where each bundle $\beta \in \mathcal{B}_N$ corresponds to a distinct core differential trail $\texttt{CDT}_i$. Then, for each node of $\mathcal{G}'$, we compute its

assigned cost as follows[9]:

$$w_N = \min_{\beta \in \mathcal{B}_N} \left\{ \frac{1 + \sum_{(N,j)\in\beta | w_j < w_N} \left(w_j / c_j^\beta \right)}{\sum_{(N,j)\in\beta | w_j < w_N} \left(1/c_j^\beta \right)} \right\}, \tag{1}$$

where for an edge (N, j) of the bundle β, c_j^β represents the cost to find a conforming message pair with difference output $j - N$ for E, and w_j is the cost assigned to the node j.

We initialize the costs of the nodes in $\mathcal{G}'$ using the shortest path in the graph, and update them iteratively until we can't find any more opportunity for improvement.

Building $\mathcal{S}$ and $\mathcal{G}$. The graph $\mathcal{G}$ is obtained from $\mathcal{G}'$ by only keeping the edges that goes from a higher to a strictly lower cost (in order to render the graph acyclic), and by only keeping for each node the outgoing edges for the bundle that minimizes the cost w_N in (1).

The set $\mathcal{S}$ is then finally deduced by harnessing all the differences corresponding to every node in $\mathcal{G}$ (one node in $\mathcal{G}$ will correspond to one difference in $\mathcal{S}$). In particular, if $\mathcal{G}'$ includes all nodes at depth at most r, then $\mathcal{S} = \{\delta_1 + \delta_2 + \cdots + \delta_r \mid \delta_1, \delta_2, \ldots, \delta_r \in \mathcal{D}\}$.

5 Application to MD5

Our techniques can easily be applied to MD5, to build an alternative chosen-prefix collision attack. We can't reach an attack complexity as low as 2^{39} (the best attack from [SSA+09]), because this would require to build a set $\mathcal{S}$ and graph $\mathcal{G}$ of size roughly 2^{50}, which is impractical. However, when the number of blocks available for the chosen-prefix collision is limited, the complexity of the best-known attack grows; for instance, the chosen-prefix collision used to create a rogue certificate was limited to 3 blocks, and this increased the complexity to

[9] In order to explain this formula, we consider that when the adversary uses a bundle β, he has to perform C_β operations to find a pair conforming to the core differential trail up to some intermediate step, and those pairs lead to an output difference $j - N$ (*i.e.* to node j) with probability p_j^β (with $p_j^\beta = C_\beta / c_j^\beta$). If none of the predetermined output differences is reached (or if the target node reached has a cost $w_j \geq w_N$), then he stays at node N and will have to still pay w_N to reach the colliding state. Thus, we have that:

$$w_N = C_\beta + \sum_{j\in\beta | w_j < w_N} \left(p_j^\beta \cdot w_j \right) + \left(1 - \sum_{j\in\beta | w_j < w_N} p_j^\beta \right) \cdot w_N$$

which leads to (1) with $c_j^\beta = C_\beta / p_j^\beta$.

2^{49}. In this scenario we can improve the currently best-known attack with our multi-block technique using a single differential trail.

We start from the single-block chosen-prefix collision attack given in [SSA+09, Section 6]: this attack uses a high probability trail for MD5 collisions, where the last steps are relaxed to allow a set $\mathcal{D}$ of size $2^{23.3}$. Therefore the birthday stage has complexity roughly $\sqrt{\pi \cdot 2^{128}/2^{23.3}} \approx 2^{53.2}$, and the near-collision block is found with a complexity of $2^{40.8}$. In our analysis, we recomputed the set $\mathcal{D}$ used by Stevens *et al.*, but we actually found a set of size $2^{24.2}$ using the same trail, with a maximum cost of $2^{26} \cdot 2^{14.8}$ (following [SSA+09], we only consider output differences with $\delta a = -2^5, \delta d = -2^5 + 2^{25}, \delta c = -2^5 \bmod 2^{20}$). Then, we extend $\mathcal{D}$ by adding the zero value and the opposite of each value, to generate $\mathcal{D}' := \mathcal{D} \cup -\mathcal{D} \cup \{0\}$. Finally, we build the set $\mathcal{S}$ and the graph $\mathcal{G}'$ corresponding to an attack with at most 2 blocks, with $\mathcal{S} := \{\delta_1 + \delta_2 \mid \delta_1, \delta_2 \in \mathcal{D}'\}$. Since the cost of the near-collision stage is negligible (at most $2 \cdot 2^{40.8}$), we do not need to use clustering, and we can just use the minimal cost paths of $\mathcal{G}'$ as the graph $\mathcal{G}$.

We find that the set $\mathcal{S}$ contains $2^{37.1}$ elements, so that the birthday stage has a complexity of roughly $\sqrt{\pi \cdot 2^{128}/2^{37.1}} \approx 2^{46.3}$. Therefore, we have a simple chosen-prefix collision attack with two near-collision blocks with complexity $2^{46.3}$, while the best previous attack with two blocks or less requires $2^{53.2}$ MD5 computations, and even the best attack with three blocks requires 2^{49} MD5 computations.

6 Application to SHA-1

For the attack on SHA-1, we directly recycle the details of the collision attack from [SBK+17]: we will use the same linear part for our successive near-collision blocks (even though the very last steps might behave slightly differently as we will explain in this section). We assume that the attacker can generate non-linear parts on the fly and can apply amortization methods just like in [SBK+17]. In order to validate this assumption, we have tried to generate a non-linear part with several random input differences from $\mathcal{S}$ and random input chaining values. In our experiments, we have successfully generated such non-linear part, and we could even make it limited to the very first SHA-1 steps. We discuss this assumption and our experiments in more details in Section 6.3.

We now explain how to apply the framework of Section 4 to a chosen-prefix collision against SHA-1. As mentioned, our attack uses the best core trail known for attacks against SHA-1, as used in previous attacks [Ste13, Ste12, SBK+17]. This allows us to have a relatively good complexity estimation for the attack, because this trail has been well studied, and a full collision attack with this trail was recently implemented. In the following we denote the complexity to find a block conforming to the trail (with an optimal output difference) as C_{block}. In the case of the recent collision attack, this cost was estimated as $C_{\text{block}} = 2^{64.7}$ SHA-1 evaluations on a GTX-970 GPU [SBK+17, Section 5.7]. In this work, we consider several hypothesis for the cost of finding near-collision blocks: an optimistic hypothesis with $C_{\text{block}} = 2^{64.7}$ (following [SBK+17]) and a conservative

hypothesis with $C_{\text{block}} = 2^{67.7}$. This parameter depends on the difficulty of linking an arbitrary input difference to the core trail, and will be discussed in more detail below.

As in the previous chosen-prefix collision attack on `SHA-1` [Ste13], we consider several variants of the core trail by changing some of the message constraints in the last steps (in particular, we flip the sign of some message bits), and we relax the last steps to reach a larger set of output difference. However, we do this more exhaustively than Stevens: he only describes a set $\mathcal{D}$ of size 192 with cost at most $1.15 \cdot C_{\text{block}}$, but we found a set of size 8768 with cost at most $8 \cdot C_{\text{block}}$, including 576 values with cost at most $1.15 \cdot C_{\text{block}}$. In particular, this directly leads to an improvement of the single-block chosen-prefix collision from [Ste13], with complexity roughly $\sqrt{\pi \cdot 2^{160}/8768} \approx 2^{74.3}$, rather than $\sqrt{\pi \cdot 2^{160}/192} \approx 2^{77}$ (ignoring some technical details of the birthday step).

More precisely, we allow the signs of the message differences to not necessarily follow local collision patterns in the last steps. Instead, we consider variants of the trail where each of those constraints is either followed or not. In addition, we fix the sign of some additional state bits to reduce the cost to reach a given output difference. Table 3 compares our message constraints with those used for the second-block of the attack from [SBK+17]. This leads to 288 differences with optimal probability ($2^{-19.17}$ in steps 61 to 79), and 288 with almost optimal probability ($2^{-19.36}$ in steps 61 to 79), as listed in Table 4. Moreover, we consider output differences whose cost is higher than the optimal cost C_{block}, up to roughly $8 \cdot C_{\text{block}}$ (we allow a probability up to 2^{-22} in steps 61 to 79).

Instead of building the corresponding set of output differences and their probability analytically, we used a heuristic approach. For each choice of the message constraints z_i in Table 3 (up to some symmetries), we generated 2^{30} intermediate states at step 60, and we computed the corresponding output differences in order to identify high probability ones. We also keep track of the differences reached with the same constraints, to build the corresponding bundles of differences. Next, we used symmetries in the set of differences to verify the consistency of the results, and to increase the precision of the heuristic probabilities. This strategy leads to a set of 8768 possible output differences, grouped in 2304 bundles. We list the output differences with (almost) optimal probability that we have identified in Table 4, with the corresponding bundles (we do not give the set with all considered differences due its large size).

Next, we build the set $\mathcal{S}$ of acceptable differences, and the graph $\mathcal{G}$ that indicates the sequences of near-collision blocks to use to cancel the differences in $\mathcal{S}$. We first build the graph $\mathcal{G}'$ as explained in Section 4.2. We use a limit on the cost of the nodes added to graph: we only consider nodes that have a path with cost at most $18 \cdot C_{\text{block}}$ in the graph $\mathcal{G}'$ (where this cost is computed with a single path, without using clustering). This yield a set of $2^{33.78}$ unique differences. After optimizing the cost with clustering, most of the nodes have a cost at most $4.5 \cdot C_{\text{block}}$, and we use a subset of the graph by bounding the cost of the near-collision stage. We describe various trade-offs in Table 5: a larger set

Stevens *et al.* constraints [SBK+17]	Our constraints
$W_{68}^{[5]} = W_{67}^{[0]} \oplus 1$	$W_{68}^{[5]} = W_{67}^{[0]} \oplus 1$
$W_{72}^{[30]} = W_{67}^{[0]} \oplus 1$	$W_{72}^{[30]} = W_{67}^{[0]} \oplus 1$
$W_{71}^{[6]} = W_{70}^{[1]} \oplus 1$	$W_{71}^{[6]} = W_{70}^{[1]} \oplus 1$
$W_{72}^{[5]} = W_{71}^{[0]} \oplus 1$	$W_{72}^{[5]} = W_{71}^{[0]} \oplus 1$
$W_{76}^{[30]} = W_{71}^{[0]} \oplus 1$	$W_{76}^{[30]} = W_{71}^{[0]} \oplus 1$
$W_{74}^{[7]} = W_{73}^{[2]} \oplus 1$	$W_{74}^{[7]} = W_{73}^{[2]} \oplus 1, W_{73}^{[2]} = z_1$
$W_{75}^{[6]} = W_{74}^{[1]} \oplus 1$	$W_{75}^{[6]} = W_{74}^{[1]} \oplus 1$
$W_{76}^{[6]} = W_{75}^{[1]} \oplus 1$	$W_{76}^{[6]} = W_{75}^{[1]} \oplus 1$
$W_{76}^{[1]} = W_{76}^{[0]} \oplus 1$	$W_{76}^{[1]} = z_2, \quad W_{76}^{[0]} = z_3$
$W_{77}^{[1]} = W_{77}^{[0]} \oplus z_1$	$W_{77}^{[1]} = z_4, \quad W_{77}^{[0]} = z_5$
$W_{77}^{[2]} = W_{77}^{[1]} \oplus 1$	$W_{77}^{[2]} = z_6$
$W_{77}^{[8]} = W_{76}^{[3]} \oplus 1, W_{76}^{[3]} = z_2$	$W_{76}^{[3]} = z_7, \quad W_{77}^{[8]} = z_8$
$W_{78}^{[0]} = W_{74}^{[7]}$	$W_{78}^{[0]} = z_9$
$W_{78}^{[3]} = z_3$	$W_{78}^{[3]} = z_{10}$
$W_{78}^{[7]} = z_4$	$W_{78}^{[7]} = z_{11}$
$W_{79}^{[2]} = z_5$	$W_{79}^{[2]} = z_{12}$
$W_{79}^{[4]} = z_6$	$W_{79}^{[4]} = z_{13}$

Table 3. Message constraints in the final steps. The z_i are fixed to `0` or `1` to define variant of the trail with distinct output differences. We use three more constraints than [SBK+17].

reduces the cost of the birthday stage, but increase the cost of the near-collision stage.

We note that the memory requirements of our attack are rather limited: one just has to store the graph, and the chains for the birthday phase. With the parameters we propose, this represents less than 1TB.

6.1 Limiting the number of near-collision blocks

The attack above is optimized to minimize the time complexity of the attack, but this can result in long paths in the graph. For instance, when starting from a random difference with cost at most $3.0 \cdot C_{\text{block}}$, a random path has on average 15.7 near-collision blocks, but the maximal length is 26 near-collision blocks. This might be impractical for some applications of chosen-prefix collision attacks, and the work needed to generate all the differential trails for the near-collision blocks might also be an issue.

Therefore, we propose an alternative attack where we limit the length of the paths in the graph $\mathcal{G}$. This result in a slightly higher complexity, but might be better in practice. More precisely, we first construct a graph with only paths of length 1, and we iteratively build graphs by increasing the length of allowed paths. Note that a given difference can often be reached by many paths of varying length, and the cost of a node decreases when allowing longer paths.

Bundle	Output difference					Proba ($-\log$)
$\mathcal{B}_1$	0xffffedea	0xffffff70	0x00000000	0x00000002	0x80000000	19.17
	0xffffedee	0xffffff70	0x00000000	0x00000002	0x80000000	19.17
	0xffffefea	0xffffff80	0x00000000	0x00000002	0x80000000	19.17
	0xffffefee	0xffffff80	0x00000000	0x00000002	0x80000000	19.17
	0xfffe5ec	0xffffff30	0x80000000	0x00000002	0x80000000	19.36
	0xfffe7ec	0xffffff40	0x80000000	0x00000002	0x80000000	19.36
$\mathcal{B}_2$	0xffffedea	0xffffff70	0x00000000	0xfffffffe	0x80000000	19.17
	0xffffedee	0xffffff70	0x00000000	0xfffffffe	0x80000000	19.17
	0xffffefea	0xffffff80	0x00000000	0xfffffffe	0x80000000	19.17
	0xffffefee	0xffffff80	0x00000000	0xfffffffe	0x80000000	19.17
	0xfffe5ec	0xffffff30	0x80000000	0xfffffffe	0x80000000	19.36
	0xfffe7ec	0xffffff40	0x80000000	0xfffffffe	0x80000000	19.36
$\mathcal{B}_3$	0xffffedea	0xffffff70	0x00000000	0x00000002	0x80000000	19.17
	0xffffedee	0xffffff70	0x00000000	0x00000002	0x80000000	19.17
	0xffffefea	0xffffff80	0x00000000	0x00000002	0x80000000	19.17
	0xffffefee	0xffffff80	0x00000000	0x00000002	0x80000000	19.17
	0xfffff5ec	0xffffffb0	0x80000000	0x00000002	0x80000000	19.36
	0xfffff7ec	0xffffffc0	0x80000000	0x00000002	0x80000000	19.36
$\mathcal{B}_4$	0xffffedea	0xffffff70	0x00000000	0xfffffffe	0x80000000	19.17
	0xffffedee	0xffffff70	0x00000000	0xfffffffe	0x80000000	19.17
	0xffffefea	0xffffff80	0x00000000	0xfffffffe	0x80000000	19.17
	0xffffefee	0xffffff80	0x00000000	0xfffffffe	0x80000000	19.17
	0xfffff5ec	0xffffffb0	0x80000000	0xfffffffe	0x80000000	19.36
	0xfffff7ec	0xffffffc0	0x80000000	0xfffffffe	0x80000000	19.36
$\mathcal{B}_5$	0xffffedaa	0xffffff6e	0x00000000	0x00000002	0x80000000	19.17
	0xffffedae	0xffffff6e	0x00000000	0x00000002	0x80000000	19.17
	0xffffefaa	0xffffff7e	0x00000000	0x00000002	0x80000000	19.17
	0xffffefae	0xffffff7e	0x00000000	0x00000002	0x80000000	19.17
	0xfffe5ac	0xffffff2e	0x80000000	0x00000002	0x80000000	19.36
	0xfffe7ac	0xffffff3e	0x80000000	0x00000002	0x80000000	19.36
$\mathcal{B}_6$	0xffffedaa	0xffffff6e	0x00000000	0xfffffffe	0x80000000	19.17
	0xffffedae	0xffffff6e	0x00000000	0xfffffffe	0x80000000	19.17
	0xffffefaa	0xffffff7e	0x00000000	0xfffffffe	0x80000000	19.17
	0xffffefae	0xffffff7e	0x00000000	0xfffffffe	0x80000000	19.17
	0xfffe5ac	0xffffff2e	0x80000000	0xfffffffe	0x80000000	19.36
	0xfffe7ac	0xffffff3e	0x80000000	0xfffffffe	0x80000000	19.36
$\mathcal{B}_7$	0xffffedaa	0xffffff6e	0x00000000	0x00000002	0x80000000	19.17
	0xffffedae	0xffffff6e	0x00000000	0x00000002	0x80000000	19.17
	0xffffefaa	0xffffff7e	0x00000000	0x00000002	0x80000000	19.17
	0xffffefae	0xffffff7e	0x00000000	0x00000002	0x80000000	19.17
	0xfffff5ac	0xffffffae	0x80000000	0x00000002	0x80000000	19.36
	0xfffff7ac	0xffffffbe	0x80000000	0x00000002	0x80000000	19.36
$\mathcal{B}_8$	0xffffedaa	0xffffff6e	0x00000000	0xfffffffe	0x80000000	19.17
	0xffffedae	0xffffff6e	0x00000000	0xfffffffe	0x80000000	19.17
	0xffffefaa	0xffffff7e	0x00000000	0xfffffffe	0x80000000	19.17
	0xffffefae	0xffffff7e	0x00000000	0xfffffffe	0x80000000	19.17
	0xfffff5ac	0xffffffae	0x80000000	0xfffffffe	0x80000000	19.36
	0xfffff7ac	0xffffffbe	0x80000000	0xfffffffe	0x80000000	19.36

Table 4. Bundles of trails with (near) optimal cost and the corresponding probability for steps 61–79. For each bundle $\mathcal{B}_i$ in the table, there are 32 related bundles where we flip some of the messages bits, that can be constructed as:
$B_0 = \{\mathcal{B}_i\}$
$B_1 = \left\{\{\beta + (2^5, 0, 0, 0, 0) \mid \beta \in \mathcal{B}\} \;\middle|\; \mathcal{B} \in B_0\right\} \cup B_0$
$B_2 = \left\{\{\beta + (2^3, 0, 0, 0, 0) \mid \beta \in \mathcal{B}\} \;\middle|\; \mathcal{B} \in B_1\right\} \cup B_1$
$B_3 = \left\{\{\beta + (2^{13}, 2^8, 0, 0, 0) \mid \beta \in \mathcal{B}\} \;\middle|\; \mathcal{B} \in B_2\right\} \cup B_2$
$B_4 = \left\{\{\beta + (2^9, 2^4, 0, 0, 0) \mid \beta \in \mathcal{B}\} \;\middle|\; \mathcal{B} \in B_3\right\} \cup B_3$
$B_5 = \left\{\{\beta + (2^6, 2, 0, 0, 0) \mid \beta \in \mathcal{B}\} \;\middle|\; \mathcal{B} \in B_4\right\} \cup B_4.$
The set used in [Ste13] corresponds to bundles $\mathcal{B}_1$ to $\mathcal{B}_4$, with extension rules B_1 to B_4. Note that most output differences appears in several bundles.

We have constructed exactly the graph with all paths of length at most 4, and all paths of length at most 8 and cost at most $3.5 \cdot C_{\text{block}}$; for larger parameters, we cannot build the full graph, but we can build an approximation by limiting the set of values as in the previous construction. We give the size of the corresponding sub-graphs in Table 6. As we can see, with 8 near-collision blocks we already have a set $\mathcal{S}$ almost as large as the set corresponding to the previous attack (cf. Table 5), so that limiting the attack to 8 blocks has a small impact on the complexity. We can even find chosen-prefix collisions with just 4 near-collision blocks with a small cost increase, using a larger threshold on the maximum cost per block. We evaluate the complexity of such attacks in detail in Table 7.

We can also study the sparseness of the values in $\mathcal{S}$ to better understand the difficulty of building the differential trails for the near collision blocks. Using the set of size $2^{29.71}$ with a limit of 8 near-collision blocks and a maximum cost of $3.0 \cdot C_{\text{block}}$, the maximum weight in the differences is 26, and the average is 15.4 (using the non-adjacent form — NAF).

6.2 Birthday Stage

For the birthday stage of the attack, we follow the approach given in [Ste13]: we consider a truncation of the `SHA-1` state by keeping bits which are likely to contain a difference, and we use the distinguished points technique of [vW99]. Parameters for the birthday step with various choice of $\mathcal{G}$ are given in Table 5; we now explain in detail the case where the maximum cost of the near-collision stage is set to $3.0 \cdot C_{\text{block}}$. First, we truncate the state to 98 bits[10] so that for a random pair of values with their difference in $\mathcal{S}$, there is a probability 0.78 that the values collide on 98 bits (this probability has been computed with the tools from [Leu12]). Reciprocally, if two truncated `SHA-1` outputs are equal, then their difference is in the set $\mathcal{S}$ with probability $2^{-31.97}$. Therefore, the birthday stage will require on average $2 \cdot 2^{31.97}$ collisions in the following function:

$$f(r) := \begin{cases} \tau(h(cv\ , \text{pad}(u))) & \text{if } u[0] = 1; \\ \tau(h(cv', \text{pad}(u))) & \text{else.} \end{cases}$$

In order to keep the cost of rerunning the trail low, we use chains of average length 2^{31} (*i.e.* a point u is distinguished when $u < 2^{98-31}$). Therefore, the expected complexity of the birthday stage is[11]:

$$T = \sqrt{\pi/2 \cdot 2^{98} \cdot 2^{32.97}} \approx 2^{65.81}\ \texttt{SHA-1}\ \text{computations}$$
$$M = 2^{65.81}/2^{31} \cdot 19\,\text{bytes} \approx 570\,\text{GB},$$

and the cost to re-run the chains to locate collisions is only $2^{32.97} \cdot 2 \cdot 2^{31} \approx 2^{64.97}$. Finally, we can evaluate the complexity of the full attacks as: $2^{65.81} + 2^{64.97} + 3.0 \cdot C_{\text{block}}$.

[10] Given by mask `0x7f800000`, `0xfffc0001`, `0x7ffff800`, `0x7fffff80`, `0x7fffffff`

[11] To store a chain, we use 40 bits for the starting point, 40 bits for the length, and $98 - 31 = 67$ bits for the output, *i.e.* 19 bytes in total.

Set $\mathcal{S}$		Birthday parameters					
Max cost	Size	Mask	Proba	# coll.	Chain len.	# chain	Attack cost
$2.0 \cdot C_{\text{block}}$	$2^{24.66}$	106 bits	0.71	$2^{30.83}$	2^{34}	$2^{34.74}$	$2^{68.74} + 2^{65.83} + 2.0 \cdot C_{\text{block}}$
$2.5 \cdot C_{\text{block}}$	$2^{28.59}$	102 bits	0.65	$2^{31.03}$	2^{32}	$2^{34.84}$	$2^{66.84} + 2^{64.03} + 2.5 \cdot C_{\text{block}}$
$3.0 \cdot C_{\text{block}}$	$2^{30.95}$	98 bits	0.76	$2^{32.44}$	2^{31}	$2^{34.55}$	$2^{65.55} + 2^{64.44} + 3.0 \cdot C_{\text{block}}$
$3.5 \cdot C_{\text{block}}$	$2^{32.70}$	98 bits	0.76	$2^{30.70}$	2^{30}	$2^{34.68}$	$2^{64.68} + 2^{61.70} + 3.5 \cdot C_{\text{block}}$
$4.0 \cdot C_{\text{block}}$	$2^{33.48}$	98 bits	0.74	$2^{29.95}$	2^{30}	$2^{34.30}$	$2^{64.30} + 2^{60.95} + 4.0 \cdot C_{\text{block}}$
$4.5 \cdot C_{\text{block}}$	$2^{33.66}$	98 bits	0.74	$2^{29.77}$	2^{30}	$2^{34.21}$	$2^{64.21} + 2^{60.77} + 4.5 \cdot C_{\text{block}}$

Table 5. Trade-offs between the cost of birthday phase and the near-collision phase.

Max Cost	1 bl.	2 bl.	3 bl.	4 bl.	5 bl.	6 bl.	7 bl.	8 bl.
$2.0 \cdot C_{\text{block}}$	$2^{9.17}$	$2^{16.30}$	$2^{19.92}$	$2^{22.05}$	$2^{23.13}$	$2^{23.95}$	$2^{24.44}$	$2^{24.55}$
$2.5 \cdot C_{\text{block}}$	$2^{10.17}$	$2^{16.62}$	$2^{21.04}$	$2^{23.76}$	$2^{25.50}$	$2^{26.58}$	$2^{27.38}$	$2^{27.92}$
$3.0 \cdot C_{\text{block}}$	$2^{10.17}$	$2^{17.10}$	$2^{21.76}$	$2^{24.66}$	$2^{26.58}$	$2^{27.95}$	$2^{28.96}$	$2^{29.71}$
$3.5 \cdot C_{\text{block}}$	$2^{12.53}$	$2^{17.89}$	$2^{22.47}$	$2^{25.62}$	$2^{27.70}$	$2^{29.18}$	$2^{30.29}$	$2^{31.22}$
$4.0 \cdot C_{\text{block}}$	$2^{12.53}$	$2^{18.60}$	$2^{22.97}$	$2^{26.34}$	$\geq 2^{28.68}$	$\geq 2^{30.35}$	$\geq 2^{31.55}$	$\geq 2^{32.15}$
$5.0 \cdot C_{\text{block}}$	$2^{12.53}$	$2^{19.65}$	$2^{24.18}$	$2^{27.44}$	$\geq 2^{29.83}$	$\geq 2^{31.64}$	$\geq 2^{32.95}$	$\geq 2^{33.04}$
$6.0 \cdot C_{\text{block}}$	$2^{12.53}$	$2^{19.79}$	$2^{24.81}$	$2^{28.26}$	$\geq 2^{30.74}$	$\geq 2^{32.55}$	$\geq 2^{33.59}$	$\geq 2^{33.59}$
$7.0 \cdot C_{\text{block}}$	$2^{13.09}$	$2^{20.37}$	$2^{25.30}$	$2^{28.82}$	$\geq 2^{31.33}$	$\geq 2^{32.93}$	$\geq 2^{33.77}$	$\geq 2^{33.77}$
$8.0 \cdot C_{\text{block}}$	$2^{13.09}$	$2^{20.62}$	$2^{25.72}$	$2^{29.27}$	$\geq 2^{31.72}$	$\geq 2^{33.09}$	$\geq 2^{33.81}$	$\geq 2^{33.81}$

Table 6. Size of the set $\mathcal{S}$ with various limits on the maximum cost and on the number of near-collision blocks. We give a lower bound when we couldn't compute the full set.

Set $\mathcal{S}$			Birthday parameters					
Max bl.	Max cost	Size	Mask	Proba	# coll.	Chain len.	# chain	Attack cost
4	$4.0 \cdot C_{\text{block}}$	$2^{26.34}$	106 bits	0.48	$2^{29.70}$	2^{33}	$2^{35.18}$	$2^{68.18} + 2^{63.70} + 4.0 \cdot C_{\text{block}}$
4	$5.0 \cdot C_{\text{block}}$	$2^{27.44}$	102 bits	0.67	$2^{32.14}$	2^{32}	$2^{35.40}$	$2^{67.40} + 2^{65.14} + 5.0 \cdot C_{\text{block}}$
4	$6.0 \cdot C_{\text{block}}$	$2^{28.26}$	102 bits	0.65	$2^{31.35}$	2^{32}	$2^{35.00}$	$2^{67.00} + 2^{64.35} + 6.0 \cdot C_{\text{block}}$
4	$7.0 \cdot C_{\text{block}}$	$2^{28.82}$	102 bits	0.64	$2^{30.82}$	2^{32}	$2^{34.74}$	$2^{66.74} + 2^{63.82} + 7.0 \cdot C_{\text{block}}$
4	$8.0 \cdot C_{\text{block}}$	$2^{29.26}$	102 bits	0.63	$2^{30.39}$	2^{32}	$2^{34.52}$	$2^{66.52} + 2^{63.39} + 8.0 \cdot C_{\text{block}}$
8	$2.0 \cdot C_{\text{block}}$	$2^{24.55}$	106 bits	0.71	$2^{30.94}$	2^{34}	$2^{34.80}$	$2^{68.80} + 2^{65.94} + 2.0 \cdot C_{\text{block}}$
8	$2.5 \cdot C_{\text{block}}$	$2^{27.92}$	102 bits	0.63	$2^{31.75}$	2^{32}	$2^{35.20}$	$2^{67.20} + 2^{64.75} + 2.5 \cdot C_{\text{block}}$
8	$3.0 \cdot C_{\text{block}}$	$2^{29.71}$	98 bits	0.73	$2^{33.73}$	2^{31}	$2^{35.19}$	$2^{66.19} + 2^{65.73} + 3.0 \cdot C_{\text{block}}$
8	$3.5 \cdot C_{\text{block}}$	$2^{31.22}$	98 bits	0.72	$2^{32.23}$	2^{30}	$2^{35.44}$	$2^{65.44} + 2^{63.23} + 3.5 \cdot C_{\text{block}}$

Table 7. Trade-offs between the cost of birthday phase and the near-collision phase with a limited number of near-collision blocks (4 or 8).

6.3 Near-Collision Stage

i	A_i	W_i
-4:	10110011001011000101111011010101	
-3:	11011000110000100010011 1un01un01	
-2:	00101010001101011101 1unnn1011n11	
-1:	00101010111000011un1nn0010n0u111	
00:	10101110001111un1u0110n1u0nn0un1n	1011un0101000101010111110-10-u0
01:	10100u101u0u10101n1nu0111n-u1-1u	nu11100111010 0011110-00-00n0110
02:	1u1u0nn010nuunnu11011uuuu0001uu1	u1nn0u0001000100010101 11---unn00
03:	u1un01uunn1u1010u0u101101nu11uu1	00uuun11110101111001110 10000-u1-
04:	n0110unnnnnnnnnnnnnn11nu1000u1n1	n0nunu00----001----0---1000uu0u1
05:	un0n011100--11001--111-1u1uu1u11	10u-1--101110-000-1100-0110n-00-
06:	1101-0-101101011110101-10nun01uu	--u--u1----------------0101uun--
07:	0nuu-00----------------0100100uu	xun-nu------------1-1----11u0u--
08:	---u01----------------0--0n010-u	----un------------------------u0
09:	0n--------------------0--1-1--0u	xn----------------------0--n-0--
10:	1--1-1------------------0---1---	x-nx-x-------------1-------uxx--
11:	-1n-----------------------0-----	--u0nn-------------------1-1-u--
12:	----0---------------------1-----	n-nxxu----------------------un---
13:	n---1----------------------------	x-uu-0----------------------u----
14:	--n----------------------------	-------------------------1-un--
15:	u-1-1--------------------------	x-nxn-----------------------n----
16:	un0-0--------------------------	----u-----------------------nu---

Table 8. Example of a `SHA-1` non-linear differential path generated for one of the differences in $\mathcal{S}$. Notations follow [DMR07]. $\delta = [-2^{17} - 2^{15} + 2^{10} - 2^8 + 2^5 + 2^6 - 2^2 + 2^0, -2^{13} + 2^{11} + 2^{10} + 2^5 - 2^3, -2^5 + 2^0, -2^4 - 2^0, 0]$, with cost $2.954 \cdot C_{\text{block}}$.

An important parameter to evaluate the cost of the attack is C_{block}, the complexity to find near-collision blocks. An optimistic hypothesis is that we can find them with same complexity as in the attack of [SBK+17], *i.e.* $C_{\text{block}} = 2^{64.7}$. As mentioned earlier, we have conducted tests to verify that one can easily find short non-linear differential paths, regardless of the input chaining difference and value, to allow for a good use of neutral bits (one path example is given in Table 8).

We note that our trails are somewhat more constrained than the trails used in the collision attack, because we have denser chaining value differences and we have a few more conditions in the last round, as seen in Table 3. This could lead to fewer degrees of freedom than in the collision attack of Stevens *et al.*, and increase the cost of finding a conforming block. In particular, this can affect the use of accelerating techinques such as neutral bits and boomerangs; boomerangs are the most powerful technique, but they require significant degrees of freedom in the path construction. Therefore, we also consider a conservative complexity estimate, where we assume that boomerangs are no longer available. Since there are three boomerangs in the trail of [SBK+17], this would give $C_{\text{block}} = 2^{67.7}$.

Our experiments show that that those assumptions are reasonable. The path given in Table 8 is about as constrained as the path used for the second block of the collision attack [SBK+17] in the first round. In particular, most condition are in the first 6 steps, and don't affect the use of neutral bits, and the same

three boomerang are available. In general we expect similar results with a few boomerangs, but this might of course vary depending on the exact chaining input diffence/value.

Finally, with the optimistic hypothesis, the best trade-off is to use a limit of $3.5 \cdot C_{\text{block}}$, for a total complexity of

$$2^{64.68} + 2^{61.70} + 3.5 \cdot C_{\text{block}} \approx 2^{66.9} \qquad (\text{using } C_{\text{block}} = 2^{64.7})$$

With the conservative hypothesis, the best trade-off is to set the limit at $2.5 \cdot C_{\text{block}}$, for a total complexity of

$$2^{66.84} + 2^{64.03} + 2.5 \cdot C_{\text{block}} \approx 2^{69.35} \qquad (\text{using } C_{\text{block}} = 2^{67.7})$$

There are other trade-offs possible between the various parameters of attack. For instance, we discussed attacks with a limited number of near-collision blocks in Section 6.1; we can now evaluate the complexity of the resulting attacks. If we limit the attack to 8 near-collision blocks, the best trade-offs give the following complexities for the optimistic and conservative hypothesis respectively:

$$2^{65.44} + 2^{63.23} + 3.5 \cdot C_{\text{block}} \approx 2^{67.2} \qquad (\text{using } C_{\text{block}} = 2^{64.7})$$
$$2^{67.20} + 2^{64.75} + 2.5 \cdot C_{\text{block}} \approx 2^{69.5} \qquad (\text{using } C_{\text{block}} = 2^{67.7})$$

Even with a limit of only 4 near-collision blocks, we have a relatively small increase of the complexity, with the following trade-offs:

$$2^{66.74} + 2^{63.82} + 7.0 \cdot C_{\text{block}} \approx 2^{68.3} \qquad (\text{using } C_{\text{block}} = 2^{64.7})$$
$$2^{68.18} + 2^{63.70} + 4.0 \cdot C_{\text{block}} \approx 2^{70.2} \qquad (\text{using } C_{\text{block}} = 2^{67.7})$$

7 Conclusion and Future Works

This work puts another nail in the `SHA-1` coffin, with almost practical chosen-prefix collisions, between five and twenty-six times more expensive than the identical-prefix collisions recently demonstrated. This shows that continued usage of `SHA-1` for certificates or for authentication of handshake messages in TLS, SSH or IKE is dangerous, and could already be abused today by a well-motivated adversary. `SHA-1` has been broken since 2004, but it is still used in many security systems; we strongly advise users to remove `SHA-1` support to avoid downgrade attacks.

More generally, our results show that, for some hash functions, chosen-prefix collision attacks are much easier than previously expected, and potentially not much harder than a normal collision search.

Our research opens several new directions. Obviously, future work will have to implement this attack to demonstrate a real chosen-prefix collision for `SHA-1`. While the computation cost of our attack is somewhat practical, `SHA-1` attacks still require a huge computation power (thousands of GPUs in order to obtain the chosen-prefix collision in a reasonable time) and a large implementation effort.

For a concrete demonstration, a good target would be to break a protocol such as TLS or IKE, or to build a rogue certificate authority.

Another research direction is to study how one can improve `SHA-1` collision attacks, not only for minimising the cost of finding a simple collision, but to improve our chosen-prefix collision search complexity. In particular, our attack requires the ability to reach many distinct output differences for the compression function. In this paper, to simplify our analysis, we only considered the differential trail from [SBK+17] because a real collision was found with this trail, and a precise complexity evaluation was conducted. However, it should be possible to increase the pool of available differences, and further reduce the total complexity, by using other (slightly more costly) differential trails.

Finally, a last direction is to evaluate how our strategy actually applies to other hash functions, such as `RIPEMD`, (reduced-round) `SHA-2`, or even others. Again, this will require a deep knowledge of the functions studied, as many details might impact the overall complexity. We can however expect that our attack strategy will be applicable mostly on classical Davies-Meier constructions inside a single-pipe Merkle-Damgård operating mode.

In order to make our work easier to verify, we are publishing some additional data and code online at: `https://github.com/Cryptosaurus/sha1-cp`.

Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments. We also thanks Paul Wouters (Red Hat) and Marc Stevens (CWI) for additionnal comments. The second author is supported by Temasek Laboratories, Singapore.

References

[BC04] Eli Biham and Rafi Chen. Near-collisions of SHA-0. In Matthew Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 290–305. Springer, Heidelberg, August 2004.

[BCJ+05] Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribault, Christophe Lemuet, and William Jalby. Collisions of SHA-0 and reduced SHA-1. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 36–57. Springer, Heidelberg, May 2005.

[BL16] Karthikeyan Bhargavan and Gaëtan Leurent. Transcript collision attacks: Breaking authentication in TLS, IKE and SSH. In *NDSS 2016*. The Internet Society, February 2016.

[Bra90] Gilles Brassard, editor. *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*. Springer, 1990.

[CJ98] Florent Chabaud and Antoine Joux. Differential collisions in SHA-0. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 56–71. Springer, Heidelberg, August 1998.

[Dam89] Ivan Damgård. A Design Principle for Hash Functions. In Brassard [Bra90], pages 416–427.

[Dam90] Ivan Damgård. A design principle for hash functions. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 416–427. Springer, Heidelberg, August 1990.

[dB94] Bert den Boer and Antoon Bosselaers. Collisions for the compressin function of MD5. In Tor Helleseth, editor, *EUROCRYPT'93*, volume 765 of *LNCS*, pages 293–304. Springer, Heidelberg, May 1994.

[DMR07] Christophe De Cannière, Florian Mendel, and Christian Rechberger. Collisions for 70-step SHA-1: On the full cost of collision search. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *SAC 2007*, volume 4876 of *LNCS*, pages 56–73. Springer, Heidelberg, August 2007.

[JP07] Antoine Joux and Thomas Peyrin. Hash functions and the (amplified) boomerang attack. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 244–263. Springer, Heidelberg, August 2007.

[Kli06] Vlastimil Klima. Tunnels in hash functions: MD5 collisions within a minute. Cryptology ePrint Archive, Report 2006/105, 2006. http://eprint.iacr.org/2006/105.

[KPS15] Pierre Karpman, Thomas Peyrin, and Marc Stevens. Practical free-start collision attacks on 76-step SHA-1. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 623–642. Springer, Heidelberg, August 2015.

[Leu12] Gaëtan Leurent. Analysis of differential attacks in ARX constructions. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 226–243. Springer, Heidelberg, December 2012.

[LPRR07] Mario Lamberger, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Second preimages for SMASH. In Masayuki Abe, editor, *CT-RSA 2007*, volume 4377 of *LNCS*, pages 101–111. Springer, Heidelberg, February 2007.

[Mer89] Ralph C. Merkle. One Way Hash Functions and DES. In Brassard [Bra90], pages 428–446.

[Mer90] Ralph C. Merkle. One way hash functions and DES. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 428–446. Springer, Heidelberg, August 1990.

[MRS15] Florian Mendel, Vincent Rijmen, and Martin Schläffer. Collision attack on 5 rounds of Grøstl. In Carlos Cid and Christian Rechberger, editors, *FSE 2014*, volume 8540 of *LNCS*, pages 509–521. Springer, Heidelberg, March 2015.

[Nat93] National Institute of Standards and Technology. FIPS 180: Secure Hash Standard, May 1993.

[Nat95] National Institute of Standards and Technology. FIPS 180-1: Secure Hash Standard, April 1995.

[Nat02] National Institute of Standards and Technology. FIPS 180-2: Secure Hash Standard, August 2002.

[Nat15] National Institute of Standards and Technology. FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, August 2015.

[Pey07] Thomas Peyrin. Cryptanalysis of Grindahl. In Kaoru Kurosawa, editor, *ASIACRYPT 2007*, volume 4833 of *LNCS*, pages 551–567. Springer, Heidelberg, December 2007.

[Riv91] Ronald L. Rivest. The MD4 message digest algorithm. In Alfred J. Menezes and Scott A. Vanstone, editors, *CRYPTO'90*, volume 537 of *LNCS*, pages 303–311. Springer, Heidelberg, August 1991.

[Riv92] Ronald L. Rivest. *RFC 1321: The MD5 Message-Digest Algorithm*. Internet Activities Board, April 1992.

[SBK+17] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 570–596. Springer, Heidelberg, August 2017.

[SKP16] Marc Stevens, Pierre Karpman, and Thomas Peyrin. Freestart collision for full SHA-1. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 459–483. Springer, Heidelberg, May 2016.

[SLdW07] Marc Stevens, Arjen K. Lenstra, and Benne de Weger. Chosen-prefix collisions for MD5 and colliding X.509 certificates for different identities. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 1–22. Springer, Heidelberg, May 2007.

[SSA+09] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen K. Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 55–69. Springer, Heidelberg, August 2009.

[Ste12] Marc Stevens. *Attacks on Hash Functions and Applications*. PhD thesis, Leiden University, June 2012.

[Ste13] Marc Stevens. New collision attacks on SHA-1 based on optimal joint local-collision analysis. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 245–261. Springer, Heidelberg, May 2013.

[vW99] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.

[WLF+05] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu. Cryptanalysis of the hash functions MD4 and RIPEMD. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 1–18. Springer, Heidelberg, May 2005.

[WY05] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 19–35. Springer, Heidelberg, May 2005.

[WYY05a] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 17–36. Springer, Heidelberg, August 2005.

[WYY05b] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin. Efficient collision search attacks on SHA-0. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 1–16. Springer, Heidelberg, August 2005.

SHA-1 is a Shambles*

First Chosen-Prefix Collision on SHA-1 and Application to the PGP Web of Trust

Gaëtan Leurent[1] and Thomas Peyrin[2,3]

[1] Inria, France
[2] Nanyang Technological University, Singapore
[3] Temasek Laboratories, Singapore
gaetan.leurent@inria.fr, thomas.peyrin@ntu.edu.sg

https://sha-mbles.github.io/

Abstract. The SHA-1 hash function was designed in 1995 and has been widely used during two decades. A theoretical collision attack was first proposed in 2004 [29], but due to its high complexity it was only implemented in practice in 2017, using a large GPU cluster [23]. More recently, an almost practical *chosen-prefix* collision attack against SHA-1 has been proposed [12]. This more powerful attack allows to build colliding messages with two arbitrary prefixes, which is much more threatening for real protocols.

In this paper, we report the first practical implementation of this attack, and its impact on real-world security with a PGP/GnuPG impersonation attack. We managed to significantly reduce the complexity of collision attacks against SHA-1: on an Nvidia GTX 970, identical-prefix collisions can now be computed with a complexity (expressed in terms of SHA-1 equivalents on this GPU) of $2^{61.2}$ rather than $2^{64.7}$, and chosen-prefix collisions with a complexity of $2^{63.4}$ rather than $2^{67.1}$. When renting cheap GPUs, this translates to a cost of US$ 11k for a collision, and US$ 45k for a chosen-prefix collision, within the means of academic researchers. Our actual attack required two months of computations using 900 Nvidia GTX 1060 GPUs (we paid US$ 75k because GPU prices were higher, and we wasted some time preparing the attack).

Therefore, the same attacks that have been practical on MD5 since 2009 are now practical on SHA-1. In particular, chosen-prefix collisions can break signature schemes and handshake security in secure channel protocols (TLS, SSH), if generated extremely quickly. We strongly advise to remove SHA-1 from those type of applications as soon as possible.

We exemplify our cryptanalysis by creating a pair of PGP/GnuPG keys with different identities, but colliding SHA-1 certificates. A SHA-1 certification of the first key can therefore be transferred to the second key, leading to an impersonation attack. This proves that SHA-1 signatures now offer virtually no security in practice. The legacy branch of GnuPG still uses SHA-1 by default for identity certifications, but after notifying the authors, the modern branch now rejects SHA-1 signatures (the issue is tracked as CVE-2019-14855).

Keywords: SHA-1 · Cryptanalysis · Chosen-prefix collision · HPC · GPU · PGP · GnuPG

*This is the full version of the paper of the same title published at USENIX Security '20.

1 Introduction

Cryptographic hash functions are present in countless security applications and protocols, used for various purposes such as building digital signature schemes, message authentication codes or password hashing functions. In the key application of digital signatures for example, hash functions are classically applied on the message before signing it, as a domain extender and also to provide security guarantees. Informally, a cryptographic hash function H is a function that maps an arbitrarily long message M to a fixed-length hash value (we denote n its bit size). *Collision resistance* is the main security property expected from a hash function: it should be hard for an adversary to compute a collision (or *identical-prefix* collision), i.e. two distinct messages M and M' that map to the same hash value $H(M) = H(M')$, where by "hard" one means not faster than the generic $2^{n/2}$ computations birthday attack.

A cryptanalyst will try to find a collision for the hash function at a reduced cost, but ad-hoc collision attacks are hard to exploit in practice, because the attacker usually has little control over the value of the actual colliding messages (in particular where the differences are inserted, which are the interesting parts when attacking a digital signature scheme). Thus, one can consider stronger variants of the collision attack more relevant in practice, such as the so-called *chosen-prefix* collision [25] or CP collision. Two message prefixes P and P' are first given as challenge to the adversary, and his goal is to compute two messages M and M' such that $H(P \| M) = H(P' \| M')$, where $\|$ denotes concatenation. With such ability, the attacker can obtain a collision with arbitrarily chosen prefixes, potentially containing meaningful information. A CP collision can also be found generically with $2^{n/2}$ computations (thus 2^{80} for a 160-bit hash function like `SHA-1`), but ad-hoc CP collision attacks are much more difficult to find than plain collision attacks, because of the uncontrolled internal differences created by the prefixes. Yet, a CP collision attack was found for the `MD5` hash function [25], eventually leading to the creation of colliding X.509 certificates, and later of a rogue Certificate Authority (CA) [27]. CP collisions have also been shown to break important internet protocols, including TLS, IKE, and SSH [1], because they allow forgeries of the handshake messages if they can be generated extremely quickly.

Largely inspired by `MD4` [19] and then `MD5` [20], `SHA-1` [16] is one the most famous cryptographic hash functions in the world, having been the NIST and de-facto worldwide hash function standard for nearly two decades. It remained a NIST standard until its deprecation in 2011 (and was forbidden for digital signatures at the end of 2013). Indeed, even though its successors `SHA-2` or `SHA-3` are believed to be secure, `SHA-1` has been broken by a theoretical collision attack in 2004 [29]. Due to its high technicality and computational complexity (originally estimated to about 2^{69} hash function calls), this attack was only implemented in practice in 2017, using a large GPU cluster [23]. Unfortunately, the `SHA-1` deprecation process has been quite slow and one can still observe many uses of `SHA-1` in the wild, because it took more than a decade to compute an actual collision, plain collisions are difficult to use directly to attack a protocol, and migration is expensive. Web browsers have recently started to reject certificates with `SHA-1` signatures, but there are still many users with older browsers, and many protocols and software that allow `SHA-1` signatures. As observed in [12], it is still possible to buy a `SHA-1` certificate from a trusted CA, many email clients accept a `SHA-1` certificate when opening a TLS connection, and `SHA-1` is also widely supported to authenticate TLS handshake messages.

Very recently, a CP collision attack against `SHA-1` has been described in [12] (but not implemented), which requires an estimated complexity between $2^{66.9}$ and $2^{69.4}$ `SHA-1` computations. It works with a two-phase strategy: given the challenge prefixes and the random differences on the internal state it will induce, the first part of the attack uses a birthday approach to limit the internal state differences to a not-too-big subset (as done in [25, 22]). From this subset, reusing basic principles of the various collision search

Table 1: Comparison of previous and new cryptanalysis results on `SHA-1`. A free-start collision is a collision of the compression function only, where the attacker has full control on all the primitive's inputs. Complexities in the table are given in terms of `SHA-1` equivalents on a GTX-970 GPU (when possible).

Function	Collision type	Complexity	Ref.
`SHA-1`	free-start collision	$2^{57.5}$	[24]
	collision	2^{69}	[29]
		$2^{64.7}$	[22, 23][a]
		$2^{61.2}$	New
	chosen-prefix collision	$2^{77.1}$	[22]
		$2^{67.1}$	[12]
		$2^{63.4}$	New

[a] Equivalent to 2^{61} `SHA-1` on CPU, $2^{64.7}$ on GPU

advances on `SHA-1`, one slowly adds successive message blocks to come closer to a collision, eventually reaching the goal after a dozen blocks. Even though these advances put the CP collisions within practical reach for very well-funded entities, it remains very expensive to conduct and also very difficult to implement as the attack contains many very technical parts.

1.1 Our Contributions

In this article, we exhibit the very first chosen-prefix collision against `SHA-1`, with a direct application to PGP/GnuPG security. Our contributions are threefold.

Complexity improvements. While the work of [12] was mostly about high-level techniques to turn a collision attack into a chosen-prefix collision attack, we have to look at the low-level details to actually implement the attack. This gave us a better understanding of the complexity of the attack, and we managed to significantly improve several parts of the attacks (See Table 1).

First, we have improved the use of degrees of freedom (neutral bits [3] and boomerangs [10]) during the search for near-collision blocks. This reduces the computational complexity for both plain and chosen-prefix collision attacks, leading to important savings: on an Nvidia GTX 970, plain collisions can now be computed with a complexity of $2^{61.2}$ rather than $2^{64.7}$ (expressed in terms of `SHA-1` equivalents on this GPU). We note that the general ideas underlying these improvements might be interesting for cryptanalysis of algorithms beyond `SHA-1`.

Second, we improved the graph-based technique of [12] to compute a chosen-prefix collision. Using a larger graph and more heuristic techniques, we can significantly reduce the complexity of a chosen-prefix collision attack, taking full advantage of the improvements on the near-collision block search. This results in a chosen-prefix collision attack with a complexity of $2^{63.4}$ rather than $2^{67.1}$.

Record computation. We have implemented the entire chosen-prefix collision attack from [12], with those improvements. This attack is extremely technical, contains many details, various steps, and requires a lot of engineering work. Performing such a large-scale computation is still quite expensive, but is accessible with an academic budget. More precisely, we can can rent cheap GPUs from providers that use gaming or mining cards in consumer-grade PCs, rather that the datacenter-grade hardware used by big

Table 2: Complexity of the attacks against SHA-1 reported in this paper on several GPUs. The complexity is given in SHA-1 equivalents (using hashcat benchmarks). For the cost evaluation we assume that one GTX 1060 GPU can be rented for a price of US$ 35/month (the two phases of the attack are easily parallelisable): https://web.archive.org/web/20191229164814/https://www.gpuserversrental.com/
To attack MD5 ‖ SHA-1, we use the multicollision attack of Joux [9] with three phases: (i) a CP collision on SHA-1, (ii) 64 collisions on SHA-1, and (iii) 2^{64} evaluations of MD5.

Function	Collision type	GPU	Time	Complexity	Cost
SHA-1	collision	GTX 970	22 years	$2^{61.2}$	
		GTX 1060	27 years	$2^{61.6}$	US$ 11k
		GTX 1080 Ti	8 years	$2^{61.6}$	
	chosen-prefix	GTX 970	99 years	$2^{63.4}$	
		GTX 1060	107 years	$2^{63.5}$	US$ 45k
		GTX 1080 Ti	34 years	$2^{63.6}$	
MD5 ‖ SHA-1	both (plain or CP)	GTX 970	1400 years	$2^{67.2}$	
		GTX 1060	1700 years	$2^{67.6}$	US$ 720k
		GTX 1080 Ti	540 years	$2^{67.6}$	

cloud providers. This gives a total cost significantly smaller than US$ 100k to compute a chosen-prefix collision. We give more detailed complexity and cost estimates in Table 2.

We have successfully run the computation over a period of two months, using 900 GPUs (Nvidia GTX 1060). Our attack uses one partial block for the birthday stage, and 9 near-collision blocks. We paid US$ 75k to rent the GPUs from GPUserversrental, but the actual price could be smaller because we lost some time tuning the attack. There is also a large variability depending on luck, and GPU rental prices fluctuate with cryptocurrency prices.

PGP/GnuPG impersonation. Finally, in order to demonstrate the practical impact of chosen-prefix collisions, we used our CP collision for a PGP/GnuPG impersonation attack. The chosen prefixes correspond to headers of two PGP identity certificates with keys of different sizes, an RSA-8192 key and an RSA-6144 key. By exploiting properties of the OpenPGP and JPEG format, we can create two public keys (and their corresponding private keys): key A with the victim name, and key B with the attacker name and picture, such that the identity certificate containing the attacker key and picture leads to the same SHA-1 hash as the identity certificate containing the victim key and name. Therefore, the attacker can request a signature of his key and picture from a third party (from the Web of Trust or from a CA) and transfer the signature to key A. The signature stays valid because of the collision, while the attacker controls key A with the name of the victim, and signed by the third party. Therefore, he can impersonate the victim and sign any document in her name.

1.2 SHA-1 Usage and Impact

Our work shows that SHA-1 is now fully and practically broken for use in digital signatures. GPU technology improvements and general computation cost decrease will further reduce the cost, making it affordable for any ill-intentioned attacker in the very near future.

SHA-1 usage has significantly decreased in the last years; in particular web browsers now reject certificates signed with SHA-1. However, SHA-1 signatures are still supported in a large number of applications. SHA-1 is the default hash function used for certifying PGP keys in the legacy branch of GnuPG (v 1.4), and those signatures were accepted by

the modern branch of GnuPG (v 2.2) before we reported our results. Many non-web TLS clients also accept `SHA-1` certificates, and `SHA-1` is still allowed for in-protocol signatures in TLS and SSH. Even if actual usage is low (a few percent), the fact that `SHA-1` is *allowed* threatens the security because a man-in-the-middle attacker can downgrade the connection to `SHA-1`. `SHA-1` is also the foundation of the GIT versioning system, and it is still in DNSSEC signatures. There are probably a lot of less known or proprietary protocols that still use `SHA-1`, but this is more difficult to evaluate.

1.3 Outline

We first recall `SHA-1` inner workings and previous cryptanalysis on this hash function in Section 2. We then provide improvements over the state-of-the-art `SHA-1` collision attacks in Section 3 and Section 4, and we describe the details of the `SHA-1` chosen-prefix collision computation in Section 5. Finally, we show a direct application of our CP collision attack with a PGP/GnuPG impersonation (together with discussions on other possible applications) in Section 6. We discuss `SHA-1` usage and the impact of our results in Section 7. Eventually, we conclude and propose future works in Section 8.

2 Preliminaries

In this section, we describe the `SHA-1` hash function (we refer to [16] for all the complete details) and summarize the previous cryptanalysis relevant to our new work.

2.1 Description of SHA-1

`SHA-1` is a 160-bit hash function that follows the well-known Merkle-Damgård [6, 15] paradigm. A padding is first applied to the message input (with message length encoded) so that we obtain a multiple of 512 bits, and this bit string is divided into blocks m_i of 512 bits each. Then, each block m_i is processed via the `SHA-1` compression function (denoted h) to update a 160-bit chaining variable (denoted cv_i) that is initialised to a constant and public initial value (denoted IV): $cv_0 = IV$. More precisely, we have $cv_{i+1} = h(cv_i, m_{i+1})$. When all blocks have eventually been processed, the last chaining variable is the final hash output.

The `SHA-1` compression function resembles other members of the `MD-SHA` family of hash functions. It uses the Davies-Meyer construction, that turns a block cipher E into a compression function: $cv_{i+1} = E_{m_{i+1}}(cv_i) + cv_i$, where $E_k(y)$ is the encryption of the plaintext y with the key k, and $+$ is a word-wise modular addition.

The internal block cipher is composed of 4 rounds of 20 steps each (for a total of 80 steps), where one step follows a generalised Feistel network. More precisely, the internal state is divided into five registers $(A_i, B_i, C_i, D_i, E_i)$ of 32-bit each and at each step, an extended message word W_i updates the registers as follows:

$$\begin{cases} A_{i+1} &= (A_i \lll 5) + f_i(B_i, C_i, D_i) + E_i + K_i + W_i \\ B_{i+1} &= A_i \\ C_{i+1} &= B_i \ggg 2 \\ D_{i+1} &= C_i \\ E_{i+1} &= D_i \end{cases}$$

where K_i are predetermined constants and f_i are boolean functions (in short: IF function for the first round, XOR for the second and fourth round, MAJ for the third round, see Table 3). Since only a single register value is updated, we can express the `SHA-1` step function using a single variable:

$$A_{i+1} = (A_i \lll 5) + f_i(A_{i-1}, A_{i-2} \ggg 2, A_{i-3} \ggg 2) + (A_{i-4} \ggg 2) + K_i + W_i.$$

Table 3: Boolean functions and constants of SHA-1

step i	$f_i(B,C,D)$	K_i
$0 \leq i < 20$	$f_{IF} = (B \wedge C) \oplus (\overline{B} \wedge D)$	0x5a827999
$20 \leq i < 40$	$_{XOR} = B \oplus C \oplus D$	0x6ed6eba1
$40 \leq i < 60$	$_{MAJ} = (B \wedge C) \oplus (B \wedge D) \oplus (C \wedge D)$	0x8fabbcdc
$60 \leq i < 80$	$_{XOR} = B \oplus C \oplus D$	0xca62c1d6

For this reason, the differential trails figures in this article will only represent A_i, the other register values at a certain point of time can be deduced directly.

The extended message words W_i are computed linearly from the incoming 512-bit message block m, the process being called message extension. One first splits m into 16 32-bit words $M_0, \ldots, M_{15}$, and then the W_i's are computed as follows:

$$W_i = \begin{cases} M_i, & \text{for } 0 \leq i \leq 15 \\ (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) \lll 1, & \text{for } 16 \leq i \leq 79 \end{cases}$$

In the rest of this article, we will use the notation $X[j]$ to refer to bit j of word X.

2.2 Previous Works

We recall here the general state-of-the-art collision search strategies that we will use for our CP collision attack. Readers only interested by the applications of our CP collision attack can skip up to Section 6. In the rest of the article, unless stated otherwise, difference will refer to the XOR difference between two bits or the bitwise XOR difference between two words.

2.2.1 Differential Trails

The first results on SHA-0 (the predecessor of SHA-1) and SHA-1 were differential in nature and obtained by building differential paths from a linearization of the compression function (we call these *linear paths*, in opposition to *non-linear paths* which have been built without linearization): in order to simplify the analysis the attacker assumes that modular additions and boolean functions f_i in the SHA-1 compression function are behaving as an XOR with regards to differential propagation. More precisely, the 32-bit modular addition is replaced by a 32-bit bitwise XOR and the f_i functions are replaced by 3-input XOR operations. These assumptions are indeed happening with a certain probability, which will basically consist of the bulk of the final attack cost. Then, in order to further simplify the analysis while trying to minimize the number of differences present in the internal state (which will in turn increase the differential trail probability and therefore improve the final attack cost), these trails are generated with a succession of so-called local collisions: small message disturbances whose influence is immediately corrected with other message differences inserted in the subsequent SHA-1 steps, while following the SHA-1 message expansion. However, in this linearization model, impossibilities might appear in the first 20 steps of SHA-1 (as in some specific cases the f_{IF} boolean function will never behave as a 3-input XOR with regards to differential propagation) and the cheapest trail candidates might not be the ones that start and end with the same difference (which is a property required to obtain directly a 1-block collision after the compression function feed-forward). This strategy could already break the collision resistance of SHA-0, but not yet for SHA-1 (due to a small rotation added in the message expansion of SHA-1, that forces disturbances to spread throughout the rounds).

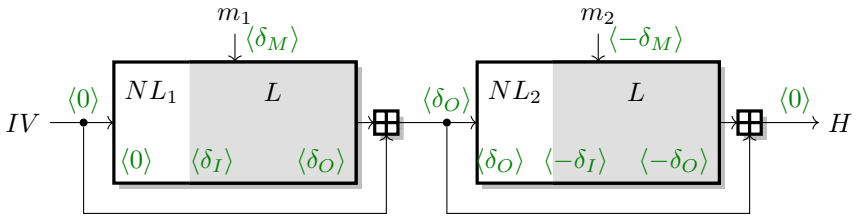


Figure 1: 2-block collision attack using a linear trail $\delta_I \overset{\delta_M}{\rightsquigarrow} \delta_O$ and two non-linear trails $0 \rightsquigarrow \delta_I$ and $\delta_O \rightsquigarrow -\delta_I$ in the first 10∼15 steps. Green values between bracket represent differences in the state.

A huge breakthrough then happened in 2005: a team of researchers [29] showed that by generating non-linear differential trails (trails generated without linearizing the `SHA-1` step function) for the first 10∼15 steps of the compression function, one could potentially connect any incoming input difference to any fixed difference δ_I at step 10∼15. This flexibility allows to remove completely the impossibility issues one could face in the first steps due to the linearization (since this part is now non-linear). Even better, taking advantage of the Davies-Meyer construction used inside the compression function, it actually permits to perform the collision attack on `SHA-1` using only two blocks containing differences, while picking the cheapest differential trail from step 10∼15 to 80. With two successive blocks using the same differential trails (just ensuring that the output difference of the two blocks have opposite signs: $0 \overset{\delta_M}{\rightsquigarrow} \delta_O$ and $\delta_O \overset{-\delta_M}{\rightsquigarrow} -\delta_O$), one can see in Figure 1 that a collision is obtained at the end of the second block because the internal state differences cancel out.

2.2.2 Improving the Efficiency of Collision Search

Once the differential trail is set the attacker must find a pair of messages that follows it. A simple strategy uses an early-abort tree exploration for the 16 first steps, taking advantage of the degrees of freedom in the message, while the remaining steps are probabilistic. More advanced amortization methods (neutral bits [3], boomerangs [11, 10] or message modification [29]) are used to control more than 16 steps. Because of this amortization, usually the first 20 or so steps (which hold with a low probability because of the non-linear trail) do not contribute to the final complexity of the attack.

Neutral bits were first introduced for the cryptanalysis of `SHA-0` [2, 3]. The idea is to find a small message modification (one or a few bits), that does not interact with necessary conditions in the differential path before a certain step x. Once a message pair following the differential path until step x is found, one can get another pair valid until step x by applying the modification. The probability that a modification is neutral until a step x can be pre-analysed before running the attack. A key observation is that any combination of two of more neutral bits until step x is likely to also be neutral until step x.

Boomerangs [10] or tunnels [11] are very similar amortization tools to neutral bits. Basically, they can be seen as neutral bits that are planned in advance. A perturbation built from one or a few local collisions (or relaxed versions) is neutral to the differential path after a few steps with a certain probability, but extra conditions are forced in the internal state and message to increase this probability. Boomerangs are generally more powerful than neutral bits (they can reach later steps than classical neutral bits), but consume more degrees of freedom. For this reason, only a few of them can be used, but their amortization gain is almost a factor 2.

Note that a lot of details have to be taken into account when using neutral bits or boomerangs, as many equations between internal state bits and message bits must be fulfilled in order for the differential path to be valid. Thus, they can only be placed at very particular bit positions and steps.

2.2.3 Chosen-prefix Collision Attacks

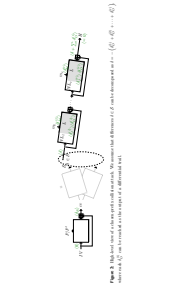
Chosen-prefix collision attacks are difficult to find for iterated hash functions such as SHA-1, because the attacker's task is then to find a collision while starting from a random difference in the internal state (due to the prefixes pair that is not controlled at all by him). This random difference prevents to use directly the collision search techniques for SHA-1 aforementioned, because the attacker has to erase this random difference somehow and the interesting differential paths are in fact a very small set, where all output differences δ_O only have a very low Hamming weight.

The first concrete application of a chosen-prefix collision attack was on MD5 [25] and this work was also the first to introduce a birthday search phase in order to partially avoid the random difference issue. The idea is to process random message blocks after the challenged prefixes, until the chaining variable difference δ belongs to a large predetermined set $\mathcal{S}$. Since the message blocks after each prefix are chosen independently, this can be done with birthday complexity $\sqrt{\pi \cdot 2^n/|\mathcal{S}|}$. Then, from that difference δ, one can reach a collision by slowly erasing the remaining unwanted difference bits by successfully applying some near-collision blocks (see Figure 2). We note that the starting difference set $\mathcal{S}$ during the birthday phase must not be too small, otherwise this phase would be too costly. Moreover, the near-collisions blocks must not be too expensive either, and this will of course depend on the cryptanalysis advancements of the compression function being studied. This two-phase strategy was applied in [22] to the full SHA-1, for a cost of $2^{77.1}$ hash calls. The improvement compared to the generic 2^{80} attack is not very large, due to the difficulty for an attacker to generate enough allowable differences that can later be erased efficiently with a near-collision block, which makes the birthday part by far the most expensive phase of the attack. In [22] a set $\mathcal{S}$ of 192 allowable differences was used, by starting from one type of near collision block and then varying the signs of the message and output differences, and also by letting some uncontrolled differences spread during the very last steps of the compression function.

This was improved in [12] by generalising for SHA-1 the set of possible differences that can be obtained for a cheap cost with a single message block, increasing the set size to 8768 elements. Another crucial improvement from [12] is the utilization of a multi-block strategy for SHA-1 that allows to further greatly increase the size of the set $\mathcal{S}$: the idea is that if an arbitrary input difference δ_R can be decomposed as $\delta_R = -\left(\delta_O^{(1)} + \delta_O^{(2)} + \cdots + \delta_O^{(r)}\right)$, where each $\delta_O^{(i)}$ can be reached as the output of a differential trail, the attacker just has to find near-collision blocks with output differences $\delta_O^{(1)}, \ldots, \delta_O^{(r)}$, where each near-collision block will cancel one of the differences $\delta_O^{(i)}$ composing δ (see Figure 2). In particular, a clustering effect appears with this multi-block strategy, which can be leveraged by the attacker to select dynamically the allowable differences at the output of each successive block, to further reduce the attack complexity. This resulted in an estimated CP collision search complexity in the range of $2^{66.9}$ to $2^{69.4}$ hash evaluations, surprisingly not much greater than that of finding a simple collision.

3 Improving SHA-1 Collision Attack

Our first contribution is an improvement of the collision attack from Eurocrypt 2013 [22] and its GPU implementation from Crypto 2017 [23]. Through better use of degrees of freedom (message modifications and boomerangs) and code improvements, we gained a



factor between 8 and 10 (depending on GPU architecture) on the time needed to find a conforming block.

Since this part of our work is very technical, we only give an overview of our results in this section. Technical details can be found in Appendix A and the corresponding code is available at https://github.com/SHA-mbles/sha1-cp.

3.1 Analysis of Previous Works

First, we observed some differences between the theoretical analysis of [22] and the practical implementation of [23]. One of the boomerangs (on bit 6 of M_6) contradicts one of the conditions used to maximize the probability of the path. Using this boomerang still improves the attack, because the gain in efficiency is larger that the loss in probability, but this affect the complexity evaluation. Similarly, one of the neutral bits used in the GPU code (on bit 11 of M_{13}) contradicts another condition in the differential path, leading to an increase in complexity of a factor $2^{0.2}$.

In our analysis, we assume that the neutral bit on bit 11 of M_{13} is not used, and that the boomerang on bit 6 of M_6 is only used for the last near-collision block, where the speed-up is most noticeable, and we have enough degrees of freedom to include all the boomerangs without difficulty. Therefore we can estimate more accurately the complexity of the previous CP attack [12] as $2^{67.1}$ `SHA-1` computations, instead of the range of $2^{66.9}$ to $2^{69.4}$ reported previously.

3.2 Additional Boomerangs

We found some additional boomerangs that can be used to speed-up the attack, on bits 4, 5, and 6 of M_{11}. Those boomerangs are not used in previous attacks because they interact badly with conditions of the differential trail, but this can be fixed by changing the last correction of the boomerangs to be a modular addition correction instead of an XOR correction.

More precisely, boomerangs are based on local collisions: an initial message difference introduces a difference in the state and another message difference cancels the state difference at a later step. In previous works, both message differences affect a single bit, so that they can be considered either as an XOR difference or as a modular difference. In this work, we only enforce a fixed modular difference for some boomerangs; depending on the value of the initial message, this difference will affect one or several bits (due to carries). Therefore, we can relax some of the conditions and make the additional boomerangs compatible with the differential path.

3.3 Precise Conditions of Neutral Bits

We also improved the rate of partial solutions generated by looking more precisely at the effect of each neutral bit. In particular, we found that some neutral bits flip with very high probability a certain condition after the step for which they are considered neutral. Therefore, these bits can be used as message modifications rather than neutral bits: instead of considering both the initial message and the message with the neutral bit applied and to test both of them at the later step, we can directly test the condition and decide which message to consider. Using this bit as message modification instead of neutral bit is more efficient, as one invalid branch in the search tree will be rightfully not explored.

In some cases, we also found that a bit that is neutral up to step i can only break some of the conditions of step i, while the rest will never be impacted. Therefore, we can test the conditions that are not affected before using that neutral bit, so as to avoid unnecessary computations. This strategy can be seen as a more precise neutral bit approach, where

Table 4: Cost of collision attacks. One collision requires on average $2^{48.5}$ A_{33}-solutions (those results include the boomerang on $M_6[8]$).
Note: we use the hashrate from hashcat, which is slightly over-optimistic (i.e. attack cost in `SHA-1` computations is overestimated).

			Collision (old)		Collision (new)		
GPU	arch	Hashrate	A_{33} rate	`SHA-1`	A_{33} rate (r)	`SHA-1`	Gain
K20x (1 GPU)	Kepler	1.7GH/s	28k/s	$2^{64.4}$	255k/s	$2^{61.2}$	9.1
GTX 970	Maxwell	3.9GH/s	59k/s	$2^{64.5}$	570k/s	$2^{61.2}$	9.6
GTX 1060	Pascal	4.0GH/s	53k/s	$2^{64.7}$	470k/s	$2^{61.6}$	8.8
GTX 1080 Ti	Pascal	12.8GH/s	170k/s	$2^{64.7}$	1500k/s	$2^{61.6}$	8.8

the attacker doesn't work step-wise, but instead condition-wise: more fine-grained filtering will lead to computation savings.

All in all, these tricks result in a better exploration of the collision search tree by cutting branches earlier. We give detailed benchmarks results and complexity estimates in Table 4, after implementing our improvements in the code of [23] (where an A_i-solution refers to an input pair that is following the differential path until word A_i inclusive).

3.4 Building Differential Trails

Following [12], we try to reuse as much as possible the previous works on `SHA-1`, and to keep our differential trail as close as possible to the attack of Stevens *et al.* [23], out of simplicity. More precisely, for each block of the collision phase, as starting point we reused exactly the same core differential path as in [23]: the difference positions in the message are the same, and the difference positions in the internal state are the same after the first 13 steps (roughly). We also tried to keep difference signs to be the same as much as possible. However, we made some modifications to the boomerangs and neutral bits as explained in the previous subsection.

The starting path skeleton is depicted in Figure 3. For each new block of the near-collision phase, we:

1. collect the incoming chaining variable and its differences and insert them inside the skeleton;
2. set the signs of the differences in the very last steps (chosen so as to minimize the final collision complexity according to the graph, see Section 4) and generate the linear system of all equations regarding the message words;
3. compute a valid non-linear differential path for the first steps;
4. generate base solutions, *i.e.* partial solutions up to A_{14}, possibly using help of neutral bits;
5. from the base solutions, search for a pair of messages that fulfils the entire differential path, using neutral bits, message modifications and boomerangs.

Steps 1 to 4 are done on CPU because they are not too computationally intensive, but step 5 runs on GPU.

We observe that in comparison with the classical collision attack [23], we have fewer degrees of freedom available in our differential paths, due to slightly more linear constraints we imposed on the late-step message bits. Moreover, for the first blocks of the near-collision phase, the attacker will have to handle a denser input difference on the chaining variable, which will render the non-linear part search a little more difficult and little more consuming in terms of freedom degrees. In any cases, we had enough degrees of freedom to find a conforming messages pair for all blocks during the attack.

i	A_i	W_i

-4:	| |	
-3:	| |	
-2:	| Incoming Chaining Variable |	
-1:	| |	
00:	|_____|	----xx----------x-
01:	????????????????????????????--	xx----------x----
02:	??????????????????????????????	x-xx-x----------xxx--
03:	??????????????????????????????	--xxxx----------x--
04:	??????????????????????????????	x-xxxx----------xx-x-
05:	????????????????????|?|????????	-x----------x----
06:	????????????????????|?|????????	--x--x----------0-0-xxx--
07:	????----------0-0??????	xxx-xx----------1-1------x-x--
08:	???x----------|--0?0--??	----xx----------x-
09:	???----------|--1?1--??	xx----------0--x----
10:	???----------|0|?---??	x-xx-x----------1-------xxx--
11:	??x----------|0|0-----	--x-xx----------111-x--
12:	----------111----	x-xxux----------xx---
13:	n----------000--	x-xx----------1u----
14:	--n----------111--	----------1-xx--
15:	u-1-1----------	x-xxx----------n----
16:	un0-0----------	----u----------nu---
17:	u--1----------	-xxnn----------n----
18:	u-u0----------	--0-n----------n-n--
19:	u----------	-xuu----------n----
20:	u-u----------	x-nux----------nnu--

Figure 3: Skeleton of starting differential path for all blocks during the near-collision phase of our CP collision attack on `SHA-1` (only the first 20 steps are depicted). The MSB's are on the right and "`-`" stands for no constraint, while the notation "`|`" on two bits vertically adjacent mean that these two bits must be equal. The other notations are similar to the ones used in [7]. This is only to give a general idea of the differential path used, as several conditions on the message and/or on the internal state are not represented here.

We also remark that the incorporation of the additional short boomerangs reduces the number of neutral bits that can be used in comparison to [23]. Yet, this was not an issue as we still had enough to keep the GPU busy (in stage 5) while the CPU was producing the base solutions (in stage 4), even though our computation cluster is composed of low range CPUs.

4 Improving `SHA-1` CP Collision Attack

In order to take advantage of the low-level improvements to collision attack techniques, we must also improve the high-level chosen-prefix collision attack.

The complexity of the birthday phase depends on the size of the set $\mathcal{S}$ of differences that can be erased from the state, therefore we need a larger set. For the near-collision phase, the complexity depends on how we combine the near-collision blocks to erase the difference in the state. We improve the graph techniques of [12] and suggest a more heuristic approach, resulting in a lower average complexity, but without a guaranteed upper bound.

4.1 Graph Construction

In order to efficiently erase the differences from the set $\mathcal{S}$, [12] uses a graph where vertices are the state difference in $\mathcal{S}$, and there is an edge between δ and δ' if $\delta' - \delta$ can be obtained as the output difference of the compression function (using a near-collision block). The birthday phase designates a starting node in the graph and we just have to follow a path leading to the zero difference, as illustrated in Figure 4. For each edge, we search for

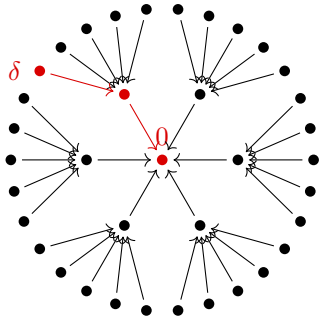


Figure 4: Graph search.

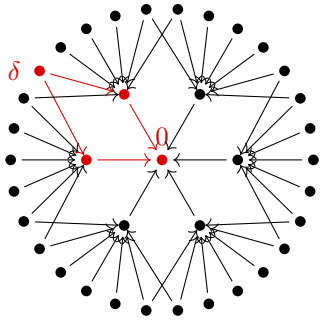


Figure 5: With clustering.

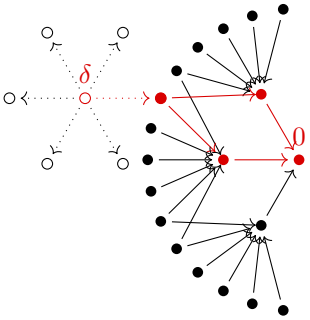


Figure 6: Bi-directional.

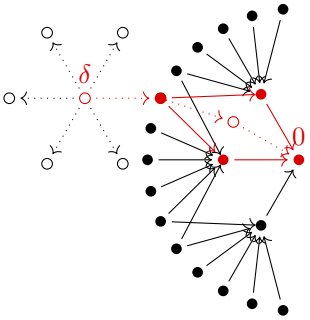


Figure 7: Implicit.

a block with the correct output difference, using near-collision search, with a cost that depends on the target difference. In the following, we denote the cost for the optimal output differences as C_{block}; it is equivalent to the cost of an identical-prefix collision.

Large graph. We started with the same approach as in [12], building a series of graphs with increasing limits on the number of blocks allowed. More precisely, we consider the set of all nodes that are reachable with a path of cost at most 24 C_{block} and up to 10 blocks. This results in a graph with $2^{36.2}$ nodes[1], which requires 2TB of storage (storing only the nodes and their cost).

Clustering. In order to minimize the complexity of the near-collision phase of the attack, [12] uses a clustering technique to exploit multiple paths in the graph (see Figure 5). Indeed, the near-collision search does not have to commit to a fixed output difference. When two output differences correspond to useful paths in the graph and are compatible with the same differential path, the attacker can run the near-collision search and stop as soon as one of them is obtained.

Concretely, let us assume we have two output differences δ_1 and δ_2 compatible with the same differential trail, that can each be reached with a cost of C_{block}. There are two different ways to erase a difference $-\delta_1 - \delta_2$ in the state:

- a block with difference δ_1, followed by a block with difference δ_2;
- a block with difference δ_2, followed by a block with difference δ_1.

If we don't decide in advance the target difference for the first block, the search is expected to reach either δ_1 or δ_2 with a cost of only 0.5 C_{block}, leading to an attack complexity of 1.5 C_{block} rather than 2 C_{block}.

[1] The largest graph suggested in [12] has size $2^{33.7}$.

Table 5: Size of the set $\mathcal{S}$ with various limits on the maximum cost and on the number of near-collision blocks ($\log_2$).

Max Cost	1 bl.	2 bl.	3 bl.	4 bl.	5 bl.	6 bl.	7 bl.	8 bl.	9 bl.	10 bl.
1 C_{block}	8.17	8.17	8.17	8.17	8.17	8.17	8.17	8.17	8.17	8.17
2 C_{block}	9.17	16.30	19.92	22.05	23.13	23.95	24.44	24.55	24.62	24.65
3 C_{block}	10.17	17.10	21.76	24.66	26.58	27.95	28.96	29.71	30.31	30.76
4 C_{block}	12.53	18.60	22.97	26.34	28.68	30.35	31.56	32.54	33.29	33.88
5 C_{block}	12.53	19.65	24.18	27.44	29.83	31.65	33.04	34.14	34.90	35.42
6 C_{block}	12.53	19.79	24.81	28.26	30.74	32.62	34.05	35.08	35.67	36.03
7 C_{block}	13.09	20.37	25.30	28.82	31.35	33.24	34.59	35.43	35.86	36.15
8 C_{block}	13.09	20.62	25.72	29.27	31.81	33.65	34.81	35.54	35.92	36.19

In our case, we initially consider nodes at distance up to 24 C_{block} and we run the clustering technique to get a better estimate of the complexity when we don't specify in advance the sequence of differences. After several weeks of computation on a machine with 48 cores and 3TB of RAM, we find that almost 90% of the nodes are actually at distance 6 C_{block} or less, as seen in Table 5.

All the differences in this set are active only on a 64-bit mask. Therefore, we use those bit positions for the birthday phase: we truncate `SHA-1` to the remaining 96 bits[2] and we generate a large number of partial collisions until one of them corresponds to a difference in the graph.

4.2 Bi-directional Graph

Since the CP collision attack is essentially a path search in a graph, we can use a bi-directional search to make the search more efficient. More precisely, when we evaluate the cost of a node, instead of just looking it up in the graph, we recompute all edges starting from the node to see if they reach the graph and compute the cost using the clustering formula. This corresponds to a bi-directional search where we pre-compute in the backwards direction the set of values that go to zero after at most 10 blocks, and during the online phase, we compute one block forward. This is illustrated by Figure 6, where black dots correspond to precomputed nodes stored in the graph, and white dots are only computed during the online phase.

This can be seen as a time-memory trade-off: we use nodes at a distance up to 11 blocks, but we only build explicitly the graph with 10 blocks. Moreover, we can use nodes that are not reachable with a single trail of cost below 24 C_{block}, and that are therefore excluded from our initial graph. Indeed, if there exists a trail such that the cost is below 24 C_{block} when removing an edge, the forward search using that edge will hit the explicit graph, and we can evaluate the distance of the node.

We can't compute exactly the size of this implicit graph, but we can evaluate it experimentally by simulating the birthday phase of the attack. We found that we need on average $2^{26.4}$ attempts before hitting the graph, which corresponds to a graph size of roughly 2^{38} (assuming that we detect being in the graph with a probability of 0.75, as was the case with the parameters of [12]).

4.3 Implicit Nodes

Following [12], we build the graph using a set $\mathcal{D}$ of 8768 potential output differences with high probability (corresponding to a cost up to 8 C_{block}). However, there are many other

[2] Given by mask `0x7f000000`, `0xfff80001`, `0x7ffff000`, `0x7ffffffc0`, `0x7fffffff`

output differences that can be useful in our attack, even if they have a lower probability: we can use a block as long as the new state difference gets closer to a collision. Therefore, during the near-collision phase, instead of keeping only blocks with an output difference corresponding to an explicit edge of the graph, we keep all blocks that follow the trail up to step 61 and we look up the new state difference in the graph (using the bi-directional strategy above). With a larger number of usable output differences, the cost of each block decreases (Figure 7).

Again, we can't compute explicitly the complexity of this attack strategy, but we can run simulations. According to our experiments with the graph described above, the average cost of the near-collision phase is only 2 C_{block}, even though most of the nodes in the graph correspond to a cost of 6 C_{block} when following edges that have been explicitly considered.

Finally, we can use this strategy to reduce the number of near-collision blocks used in the attack. In practice, we observed that most of the nodes in our graph can actually be reached with fewer than 11 blocks. In particular, when using output differences that do not correspond to edges of the graph, we often reach an output difference that can be erased with fewer blocks than expected, in particular for the first near-collision blocks.

5 Chosen-Prefix Collision Computation

Even though we managed to reduce the cost of the chosen-prefix collision for `SHA-1` to only $2^{63.7}$ `SHA-1` evaluations, performing such a large-scale computation remains very expensive. We show that it can be computed with an academic budget, for a total cost much lower than US$ 100k.

5.1 Attack Parameters

Using more blocks for the near-collision blocks part of the attack leads to improved attack complexity, but one can observe that the improvement becomes marginal as the number of block increases. Besides, in order to allow for a practical real-life use of our chosen-prefix collision (see next section), we had to enforce a limit on the total number of blocks.

Using the idea described in the previous section, we have the following parameters for the attack:

- We use a limit of at most 11 blocks, but we aim for 10 blocks at most for the attack (to fit in a 6144-bit key, see next section);
- The graph $\mathcal{G}$ has size roughly 2^{38}, but it is not computed explicitly;
- The birthday stage is a parallel collision search algorithm (using the distinguished points technique of [28]) with a mask of 96 bits, and we need about $2^{26.4}$ partial collisions on those 96 bits. Therefore the expected complexity of the birthday phase is $\sqrt{\pi 2^{96} 2^{26.4}} \approx 2^{62}$;
- We use chains (consecutive iterations of the function from a starting point during the distinguished points technique) of length 2^{28}, resulting in a data complexity of $1/2$ TB to store 2^{34} chains;
- We expect a cost of 2 C_{block} for the near-collision phase.

In hindsight, we could have use longer chains to reduce the data storage, and to make sorting the data on our cluster easier. We could also have aimed for a lower number of blocks: our collision used only 9 blocks, and we could probably reach 8 or 7 without much impact on the complexity.

Complexity estimate. Overall, for the attack parameters chosen, the birthday part costs about $2^{62.05}$ `SHA-1` computations, while the near-collision part is expected to require 1 C_{block} for the last block, and 1 C_{block} in total for the previous blocks.

As explained in Section A.1, we use the boomerang on $M_6[8]$ for the last block, so that the expected time to find a conforming block can be estimated directly from the figures of Table 4 as $C_{\text{block}} = 2^{48.5}/r$. For the intermediate blocks, we don't use this boomerang, so the rate is reduced to $r/1.9$ but we only require $2^{48.08}$ A_{33}-solutions for one C_{block}. Our simulations show that the total cost for all intermediate blocks is roughly one C_{block}, therefore it will take time $C_{\text{block}} = 1.9 \cdot 2^{48.08}/r$. Finally, we can estimate the total attack time as

$$2^{62.05} \cdot h + \frac{2^{48.5} + 1.9 \cdot 2^{48.08}}{r},$$

with r the A_{33}-solution rate (from Table 4), and h the hash-rate for the birthday phase (from Section 5.3). We give concrete complexity estimates on several GPUs in Table 2. Our chosen-prefix collision attack is roughly four time as expensive as an identical-prefix attack.

5.2 A GPU Cluster

We originally estimated that our attack would cost around US$ 160k by renting GPUs from a cloud provider such as Amazon or Google (using spot or preemptible prices). However, since our computations do not need much communication between the GPUs, nor fancy inter-GPU task scheduling, we can consider renting cheaper GPUs from providers that use gaming or mining cards in consumer-grade PCs, rather that the datacenter-grade hardware used by big cloud providers. Services like `gpuserversrental.com` rent GTX 1060 or GTX 1080 GPUs for a price below 5 cents per month per CUDA core; which would give a total cost around US$ 75k to compute a chosen-prefix collision.

After some cost analysis, we have concluded that GTX 1060 GPUs offered a very good hashrate/cost ratio at the time of the chosen-prefix collision computation. GPU prices vary significantly depending on cryptocurrency prices, but at the time of writing, a GTX 1060 can be rented for about US$ 35 per month[3]. Our attack requires about 107 GPU-years using GTX 1060, which gives an estimated cost of $107 \times 35 \times 12 \simeq$ US$ 45k to compute a chosen-prefix collision for `SHA-1`.

Our cluster was made of 150 machines with 6 GPU each (with a mix of GTX 1060 3G, and GTX 1060 6G), and one master node with two 2TB hard drives in a RAID configuration. The master node had a Core i7 CPU, but the GPU nodes had low-end Pentium or Celeron CPU with two cores. Each machine ran Ubuntu Linux, but there was no cluster management software installed (we used `clush` to run commands on all the nodes). We negotiated a price of US$ 37.8k per month (higher than current prices), and used the cluster for two months.

Cost analysis. We paid US$ 75.6k for our computation, but the cost could be as low as US$ 50k with currently lower GPU prices and less idle time. With the same methods, computing an identical-prefix `SHA-1` collision would cost only about US$ 11k. This is clearly within reach of reasonable attackers.

Of course the underlying weakness of `SHA-1` has always been present, even if it was not public (and maybe not discovered). We estimate that a PS3 cluster (as used by Stevens et al. [27], and as deployed by the US army[4]) could have implemented this attack for a cost of a few million dollars in 2010, when SHA-1 was still the most widely used hash function. This underlines that the deprection process of SHA-1 should have been much faster after the publication of the first theoretical collision attack in 2004.

[3] More precisely, US$ 209 per month for 6 GTX 1060 3GB: https://web.archive.org/web/20191229164814/https://www.gpuserversrental.com/

[4] https://phys.org/news/2010-12-air-playstation-3s-supercomputer.html

Table 6: Timeline of the birthday phase.

Date	Event	Complexity	# collisions
July 25	Starting cluster setup		
July 27	Computation started		
August 14	Step 2 unsuccessful	$2^{61.9}$	$2^{25.8}$
August 20	Step 2 unsuccessful	$2^{62.4}$	$2^{26.6}$
August 24	Step 2 unsuccessful	$2^{62.6}$	$2^{27.1}$
August 30	Step 2 successful!	$2^{62.9}$	$2^{27.7}$

Looking at the future, this attack will get even cheaper as computation costs decrease. Following Moore's law (that seems to be still valid for GPU[5]), we estimate that it should cost less than US$ 10k to generate a `SHA-1` chosen-prefix collision by 2025.

5.3 Birthday Phase

In order to simplify the implementation, we implemented the birthday phase with two distinct steps: in the first step, each GPU computes independently a series of chains, and in the second step we gather all the results, sort them to find collisions in the end-points, and re-run the chain to locate the collisions. Our implementation runs at a speed of $h = 3.5$GH/s on GTX 1060 GPUs (respectively 3.2 GH/s on GTX 970 and 11 GH/s on GTX 1080 Ti). This is somewhat lower than the hashcat benchmarks reported in Table 2 because hashcat can skip some parts of `SHA-1`, and we have to keep two `SHA-1` states in the registers to implement the birthday phase. Every time we run the second step, we then search the collisions in the graph, to determine whether we have reached a useful starting point (this is run on a separate machine with at least 1TB or RAM, and we let the cluster restart the first step in the meantime).

As shown in Table 6, we ran step 2 four times, and we have been quite unlucky in the birthday phase, only succeeded after finding $2^{27.7}$ collisions, rather than the estimated $2^{26.4}$. It took us 34 days to compute those chains, which corresponds to a hashrate 2.9 TH/s for our cluster (including downtime).

Interestingly, we got slightly fewer collisions than expected (after a given number of chains): we expected to compute $\sqrt{\pi 2^{96} C}$ `SHA-1` to find C partial collisions, but our analysis is off by a factor roughly $2^{0.2}$. Given the small magnitude of the error, we didn't investigate further, but it could be due to an unknown bug in our code, or an issue in the analysis (such as a failed independence assumption, or an issue with chains that reach a cycle).

5.4 Near-collision Phase

The near-collision phase is very technical and very complex. Every time a block is found, we have to prepare the search for the next block. This first requires to traverse the graph $\mathcal{G}$ to find the parameters for the next block: we have different constraints in the last steps depending on which output differences are desired. Then, we had to generate a new non-linear part for the early steps, as explained in Section 3.4. We used tools similar to [7], which take a lot of parametrization and trial-and-error to have a proper non-linear part that fits nicely with the core differential path. Finally, some testing and configuration of the GPU code was then required to check how neutral bits and boomerangs behaved in this new configuration. In particular, there were usually some adjustments to make in the

[5] https://blogs.nvidia.com/blog/2017/05/10/nvidia-accelerates-ai-launches-volta-dgx-workstation-robot-simulator-more/

Table 7: Timeline of the near-collision phase. C_{block} corresponds to $2^{19.17}$ A_{61}-solutions, excepted for the last block where the use of an extra boomerang increases it to $2^{19.58}$

Date	Event	$\#A_{61}$-sol	Complexity
September 07	Block 1 found[a]	2^{16}	0.11 C_{block}
September 09	Block 2 found	$2^{13.5}$	0.02 C_{block}
September 13	Block 3 found	$2^{16.9}$	0.21 C_{block}
September 14	Block 4 found	$2^{10.8}$	0.003 C_{block}
September 16	Block 5 found	$2^{15.5}$	0.08 C_{block}
September 18	Block 6 found	$2^{15.5}$	0.08 C_{block}
September 20	Block 7 found	2^{16}	0.11 C_{block}
September 21	Block 8 found	$2^{14.5}$	0.04 C_{block}
September 27	Block 9 found[b]	$2^{18.2}$	0.38 C_{block}

[a] Two solutions found
[b] Using the $M_6[8]$ boomerang

GPU code for the more complex conditions in the path that involve several bit positions. The entire preparation process would have potentially to be performed again in case of any issue detected with the path found.

This was automated to some extent, but still took between a few hours and a few days of manual work to prepare for each block (it took more time for the first blocks because there are more constraints to build the path, and we were more experienced for the later blocks). Unfortunately, this means that the GPU cluster was not doing useful work during this time. We remark that our attack could have cost less if we had fully automated the entire cryptanalysis process, or if we had improved the search algorithm for the non-linear part of the differential path. This is definitely not impossible to achieve, but it would require a lot of tedious work.

For the last block, we started the computation without the boomerang on $M_6[8]$, and modified the path and the code after one day to include it. As explained in Section 3, this extra boomerang reduces the quality of A_{61}-solutions, so that we need $4/3$ times the number of solutions ($2^{19.58}$ instead of $2^{19.17}$), but it almost doubles the production rate of these solutions. In total, this reduces the computation time by a factor $1.9/4/3 \approx 1.4$.

As expected, intermediate blocks cost much less than C_{block} (the cost of a block with a pre-determined output difference) because we can target a large number of output differences. Only the last block is expected to cost C_{block}. However, we were quite lucky in this phase of attack, because we found all the blocks after only 0.9 C_{block}, rather than the estimated 2 C_{block}. In particular, the last block was found after only $2^{18.2}$ A_{61}-solutions (0.38 C_{block}), instead of the expected $2^{19.58}$.

A timeline of the near-collision phase is given in Table 7, and the chosen-prefix collision is given in Appendix B (with the actual messages in Figure 13 and the intermediate differences in Table 9) .

5.5 Resources Used

A quick overview of the resources used for each part is given in Table 8. If we evaluate the total useful GPU time spent for the attack, we have roughly:

- 78 years for the birthday phase
- 25 years for blocks 1 to 9
- 10 years for the last block

Table 8: Resources used for the attack

Phase	Step	Main resource	Repetitions	Wall time
Setup	Preparation of the graph	CPU and RAM		$\approx$ 1 month
Birthday	Computing chains	GPU		34 days
	Sorting chains	Hard drive	4 $\times$	$\approx$ 1 day
	Locating collisions	GPU	4 $\times$	$< 1/2$ day
	Searching in graph	RAM	4 $\times$	$< 1/2$ day
Blocks	Building trail & code	Human Time	9 $\times$	$\approx$ 1 day
	Finding block	GPU	8 $\times$	3 hours – 3 days
	Checking results in graph	RAM	8 $\times$	$< 1/2$ hour
	Finding last block	GPU	1 $\times$	6 days

This means that roughly 75% of our GPU time was useful. If we convert the attack time to `SHA-1` evaluations, we arrive at a total of $2^{63.6}$, which is quite close to the estimate of $2^{63.5}$ given in Table 2.

6 Application to PGP Web of Trust

Our demonstration of a chosen-prefix collision targets the PGP/GnuPG Web of Trust. This trust model relies on users signing each other's identity certificate, instead of using a central PKI. For compatibility reasons the legacy branch of GnuPG (version 1.4) still uses `SHA-1` by default to sign identity certificates.

Therefore, we can impersonate a user using a `SHA-1` chosen-prefix collision to forge the signature.More precisely, our goal is to create two PGP keys with different UserIDs, so that key B is a legitimate key for Bob (to be signed by the Web of Trust), but the signature can be transferred to key A which is a forged key with Alice's ID. This will succeed if the hash values of the identity certificates collide, as in previous attacks against X.509 `MD5`-based certificates [25, 27]. Moreover, due to details of the PGP/GnuPG certificate structure, our attack can reuse a single collision to target arbitrary users Alice and Bob: for each victim, the attacker only needs to create a new key embedding the collision, and to collect a `SHA-1` signature. This is arguably the first practical attack against a real world security application using weaknesses of `SHA-1`.

We recall that a chosen-prefix collision attack works as follows: given two arbitrary prefixes P and P', an attacker can generate two messages M and M' such that $H(P \| M) = H(P' \| M')$. Note that in classical iterated hash functions such as `SHA-1`, given an arbitrary suffix X, we still have $H(P \| M \| X) = H(P' \| M' \| X)$.

6.1 Exploiting a Chosen-prefix Collision

We now focus on the identity certificates that will be hashed and signed. Following RFC 4880 [5], the hash computation done during certificate signing receives the public key packet, then a UserID or user attribute packet, and finally a signature packet and a trailer. The idea of the attack is to build two public keys of different sizes, so that the remaining fields to be signed are misaligned, and we can hide the UserID of key A in another field of key B. Following RFC 4880, the signature packet is protected by a length value at the beginning *and at the end*, so that we have to use the same signature packet in key A and key B (we cannot stuff data in the hashed subpacket). Therefore, we can only play with the UserID and/or user attribute packets. Still, a user attribute packet with a JPEG image gives us enough freedom to build colliding certificates, because typical JPEG readers

ignore any bytes after the End of Image marker (`ff d9`). This gives us some freedom to stuff arbitrary data in the certificate.

More precisely, we build keys A and B as follows. Key A contains an 8192-bit RSA public key, and a UserID field corresponding to Alice. On the other hand, key B contains a 6144-bit RSA public key, the UserID of Bob and a JPEG image. Therefore, when Bob gets a certification signature of his key, the signer will sign two certificates: one containing his public key and UserID, and another one containing the public key and the image. The public keys A and B and the image are crafted in such a way to generate a collision between the certificates with the key A and Alice's UserID, and the certificate with key B and the image.

6.1.1 Content of Identity Certificates

Figure 8 shows a template of the values included in the identity certificate: those values are hashed when signing a key, and we want the two hashes to collide. In this example, the UserID field of key A contains "`Alice <alice@example.com>`", and the image in key B is a valid JPEG image that will be padded with junk data after the End of Image marker. The real JPEG file is 181 bytes long[6] (from `ff d8` to `ff d9`), and it is padded with 81 bytes, so that the file included in the key is 262 bytes long (here the padding includes 46 bytes corresponding to the end of the modulus of key A, 5 bytes corresponding to the exponent of key A, and 30 bytes corresponding to Alice's UserID).

In Figure 8, we use the following symbols:

`01` Bytes with a fixed value are fixed by the specifications, or chosen in advance by the attacker (length of fields, UserID, user attribute, ...)
`??` Represent bytes that are determined by the chosen-prefix collision algorithm (the messages M and M' to generate a collision)
`!!` Represent bytes that are selected after finding the collision, to generate an RSA modulus with known prime factors
`..` Represent bytes that are copied from the other certificate
`**` Represent time-stamps chosen by the attacker
`$$` Represent the time-stamp chosen by the signer

Underlined values correspond to packet headers (type and length).

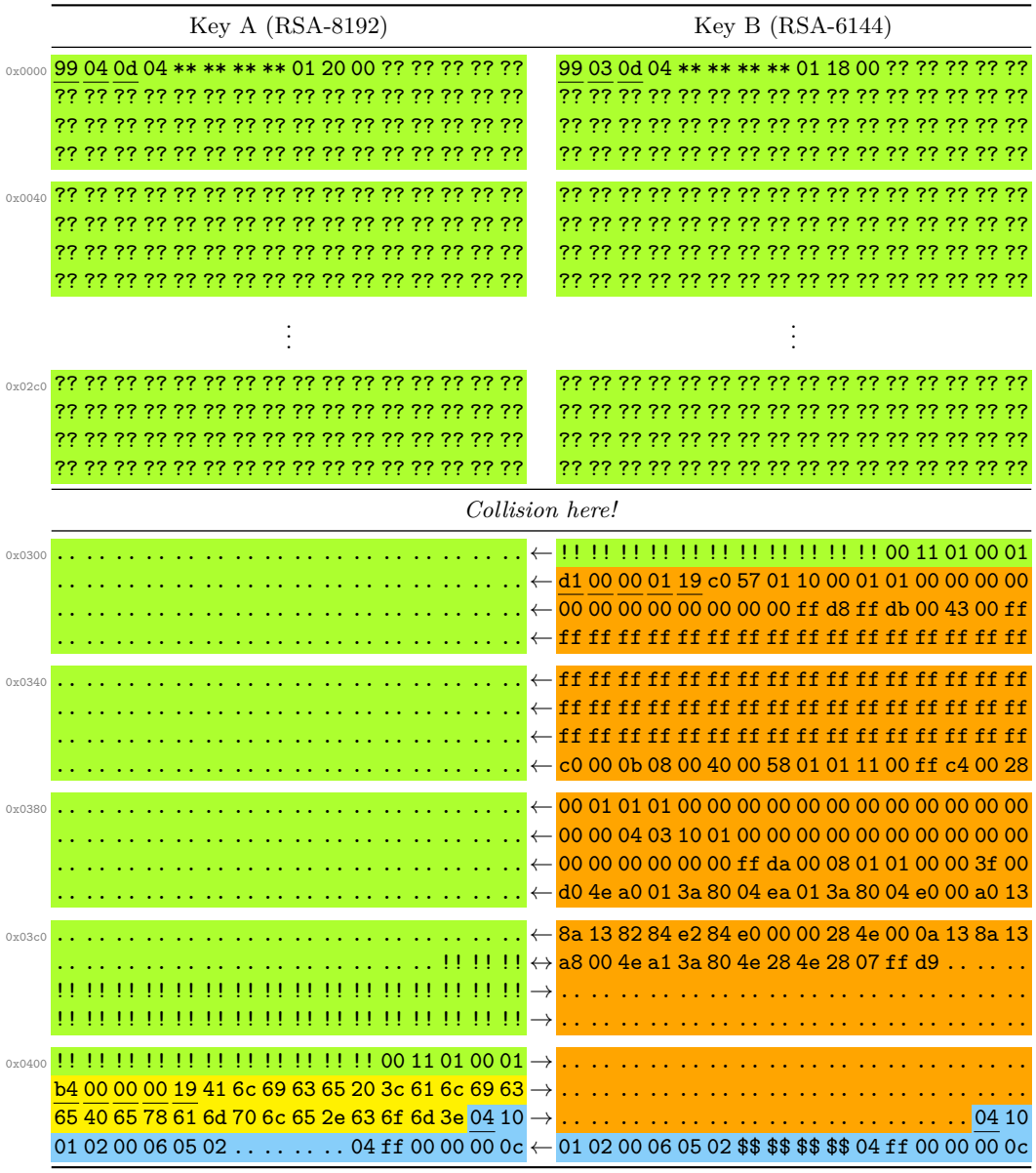
6.1.2 Attack Procedure

To carry out the attack, we have to perform the following steps:

1. Build a chosen-prefix collision with prefixes "`99 04 0d 04 ** ** ** ** 01 20 00`" and "`99 03 0d 04 ** ** ** ** 01 18 00`", after filling the `**` with two arbitrary time-stamps. The chosen-prefix collision must have at most 10 near-collision blocks. This determines the `??` bytes of the keys.
2. Choose a tiny JPEG image to include in key B (fixed orange bytes), and an arbitrary UserID to include in key A (fixed yellow bytes)
3. Select "`!!`" bytes in B to obtain a modulus with known factors
4. Select "`!!`" bytes in A to obtain a modulus with known factors
5. Generate key B with the modulus and the padded JPEG. Ask for a signature of the key.
6. Copy the signature to key A.

We point out that the chosen-prefix collision is computed *before* choosing the UserIDs and images that will be used in the attack. Therefore, a single CPC can be reused to

[6] Building a JPEG image smaller than 256 bytes is not easy, but it is possible



```
                Key A (RSA-8192)                                  Key B (RSA-6144)
0x0000 99 04 0d 04 ** ** ** ** 01 20 00 ?? ?? ?? ?? ??    99 03 0d 04 ** ** ** ** 01 18 00 ?? ?? ?? ?? ??
       ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??    ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
       ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??    ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
       ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??    ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??

0x0040 ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??    ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
       ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??    ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
       ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??    ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
       ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??    ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??

                          ⋮                                                 ⋮

0x02c0 ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??    ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
       ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??    ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
       ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??    ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
       ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??    ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
                                       Collision here!
0x0300 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ← !! !! !! !! !! !! !! !! !! !! !! 00 11 01 00 01
       .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ← d1 00 00 01 19 c0 57 01 10 00 01 01 00 00 00 00
       .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ← 00 00 00 00 00 00 00 00 ff d8 ff db 00 43 00 ff
       .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ← ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff

0x0340 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ← ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
       .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ← ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
       .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ← ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
       .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ← c0 00 0b 08 00 40 00 58 01 01 11 00 ff c4 00 28

0x0380 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ← 00 01 01 01 00 00 00 00 00 00 00 00 00 00 00 00
       .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ← 00 00 04 03 10 01 00 00 00 00 00 00 00 00 00 00
       .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ← 00 00 00 00 00 00 ff da 00 08 01 01 00 00 3f 00
       .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ← d0 4e a0 01 3a 80 04 ea 01 3a 80 04 e0 00 a0 13

0x03c0 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ← 8a 13 82 84 e2 84 e0 00 00 28 4e 00 0a 13 8a 13
       .. .. .. .. .. .. .. .. .. .. .. .. .. !! !! !! ↔ a8 00 4e a1 3a 80 4e 28 4e 28 07 ff d9 .. .. ..
       !! !! !! !! !! !! !! !! !! !! !! !! !! !! !! !! → .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
       !! !! !! !! !! !! !! !! !! !! !! !! !! !! !! !! → .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..

0x0400 !! !! !! !! !! !! !! !! !! !! !! 00 11 01 00 01 → .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
       b4 00 00 00 19 41 6c 69 63 65 20 3c 61 6c 69 63 → .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
       65 40 65 78 61 6d 70 6c 65 2e 63 6f 6d 3e 04 10 → .. .. .. .. .. .. .. .. .. .. .. .. .. .. 04 10
       01 02 00 06 05 02 .. .. .. .. 04 ff 00 00 00 0c ← 01 02 00 06 05 02 $$ $$ $$ $$ 04 ff 00 00 00 0c
```

Figure 8: Construction of colliding OpenPGP identity certificates. The colour corresponds to the packets hashed when computing the signature: first, the 🟩 public key packet (with header), then the 🟨 UserID or 🟧 user attribute, and finally the 🟦 signature packet and trailer. Arrows show when a value is chosen in one key and copied to the other.

attack many different victims. This contrasts with attacks on X.509 certificates [25, 27], where the identifier is hashed before the public key.

In order to build the modulus (steps 3 and 4 above), we use the same strategy as in previous works [25, 27]. More precisely, the high order bits are fixed by previous steps, and the low-order bits can be chosen freely. Therefore we have to find a modulus in an interval $[A, B]$ with a known factorisation. We select a random prime P (in the order of $B - A$), and we compute $Q = \lfloor B/P \rfloor$. If Q is a prime, we use $P * Q$ as the modulus: we have $A \leq P * Q \leq B$ when $P \leq B - A + 1$. This takes a few minutes in practice.

We note that the factors of the modulus are unbalanced. With the template of Figure 8, we expect factors of 88 bits and 6056 bits for Key B, and 368 bits and 7824 bits for key A. In practice we managed to find a CP collision with fewer blocks than in Figure 8, so that key B actually has factors of 1112 bits and 5032 bits. This makes both keys hard to factor. As mentioned in [14], it is possible to find modulus with somewhat larger factors using more advanced techniques.

6.1.3 Example Keys

We show an example of a pair of keys generated with this procedure in Figures 11 and 12 from the Appendix. The keys can be examined with `pgpdump -i` to see that they include the same signature. The files can be directly downloaded from these URLs:

Key A: https://SHA-mbles.github.io/alice.asc

Key B: https://SHA-mbles.github.io/bob.asc

In our demonstration, we chose a time-stamp far in the future to avoid malicious usage of our collision. However, an attacker that can repeat our work will obviously use a valid time-stamp.

6.1.4 Attack Variant

We also found an alternative attack, exploiting the PGP key format in a slightly different way, where key B contains a short public key followed by a JPEG image. We would consider both the public key and the image as the prefix, and stuff the CPC blocks inside the image (after the EOI marker). This variant leaves a smaller space for the CPC blocks, but the advantage is that key A is less suspicious because it doesn't need to contain a valid JPEG file inside the modulus (the modulus is really made of random-looking blocks). On the other hand, this variant requires to compute a new CPC for each key B.

Other variants might also be possible.

6.2 Impact

As explained in Section 7.1, the "classic" branch of GnuPG (v1.4) uses `SHA-1` by default for identity certifications, and there is still a non-negligible number of keys signed with `SHA-1`. Before our attack was disclosed, `SHA-1` signatures were also accepted by the "modern" branch of GnuPG (v2.2). This made the attack usable in practice.

In addition, a single CPC can be reused to attack many different victims, so that the cost of the CPC is just a one-off cost. Given our cost estimation around US$ 50k, this is well within reach of strong adversaries.

7 `SHA-1` Usage and Disclosure

`SHA-1` is still used in a surprising number of security applications. It is supported in many secure channel protocols (TLS, SSH), and remains actually used for some fraction of the

connections. It is also used for PGP identity certifications, and it is the foundation of GIT versioning system. We expect there are also an important number of proprietary systems using SHA-1, but getting actual data on this is difficult.

Collisions and chosen-prefix collisions do not threaten all those usages (in particular HMAC-SHA-1 seems relatively safe), but there are several settings that are directly affected by chosen-prefix collisions:

- PGP identities can be impersonated if trusted third parties sign identity certificates with SHA-1 (see 7.1)
- X.509 certificates could be broken if some CAs issue SHA-1 certificates with predictable serial numbers (see 7.2)
- TLS and SSH connections using SHA-1 signatures to authenticate the handshake could be attacked with the SLOTH attack [1] if the CP collision can be generated extremely quickly (see 7.3 and 7.4)

We stress that when a protocol supports several hash functions, those attacks are possible as long as SHA-1 is *supported* by implementations, even if it is not selected during normal use. A man-in-the-middle attacker will just force the parties to use SHA-1.

More generally, as cryptographers, we recommend to deprecate SHA-1 everywhere, even when there is no direct evidence that this weaknesses can be exploited. SHA-1 has been broken regarding collision resistance for 15 years, and there are better alternatives available, well-studied, and standardized (SHA-2 [17], SHA-3 [18]). There is no good reason to use SHA-1 in modern security software. Attacks only get better over time, and the goal of the cryptanalysis effort is to warn users so that they can deprecate algorithms *before* the attacks get practical.

As a stopgap measure, the collision-detection library of Stevens and Shumow [26] can be used to detect attack attempts (it successfully detects our attack).

Responsible disclosure. We have tried to contact the authors of affected software before announcing this attack, but due to limited resources, we could not notify everyone. We detail below the main affected products, some of the responses we received, and countermeasures deployed at the time of writing. More up to date information will be available on the website of the attack: https://sha-mbles.github.io.

7.1 SHA-1 Usage in GnuPG

There are currently two supported branches of GnuPG: GnuPGv1 is the "legacy" (or "classic") branch, and GnuPGv2 is the "modern" branch. The first version of GnuPGv2 dates back to 2006, and the "legacy" branch is no longer recommended, but the transition took a long time. In particular, GnuPGv1 was still the default version in Fedora 29 (released in October 2018), and in Ubuntu 16.04 LTS (which is supported until April 2021).

GnuPG supports many different algorithms, including SHA-1. Moreover, SHA-1 is the default algorithm for identity certification in GnuPGv1. This is why we targeted PGP in our demonstration of chosen-prefix collisions. After we disclosed our results to the GnuPG team, SHA-1 signatures have been deprecated in the GnuPGv2 branch.

We have first discussed this attack with the GnuPG developers the 9th of May 2019 and eventually informed them of the newly found chosen-prefix collision the 1st of October 2019. The issue is tracked with CVE number CVE-2019-14855. A countermeasure has been implemented in commit edc36f5, included in GnuPG version 2.2.18 (released on the 25th of November 2019): SHA-1-based identity signatures created after 2019-01-19 are now considered invalid.

Web of Trust. The original trust model of PGP was the Web of Trust. Instead of using a central PKI, users sign each other's keys to attest of their identity (*e.g.* when attending a key signing party), and trust such certificates from third parties. A scan of the PGP Web of Trust (i.e. identity certifications on public keyservers) shows that roughly 1% of the identity certifications issued in 2019 use `SHA-1`. This probably corresponds to usage of GnuPGv1 with the default settings, and would make our attack feasible.

However the Web of Trust does not seem to be widely used anymore. In particular, after the poisoning attack at the end of June 2019 [**?**], GnuPG 2.2.17 and later do not import identity certificates from public keyservers by default. A major usage of GnuPG is now to authenticate software packages in Linux, but this typically relies on directly trusting the relevant keys without third parties.

CAcert. CAcert (`http://cacert.org/`) is one of the main CAs for PGP keys, and they still use `SHA-1` to sign user keys. We have first contacted them by email on December 14th, and got an answer on January 6th acknowledging this issue. They are now planning a switch to a secure hash function for key certification.

We note that our attack is not directly applicable because CAcert does not sign JPEG images in PGP keys, but using `SHA-1` signature is nonetheless an important security risk.

7.2 `SHA-1` Usage in X.509 Certificates

The CA/Browser Forum decided to sunset `SHA-1` in October 2014, and its members are not supposed to issue `SHA-1` certificates after 2016. Web browsers have enforced similar rules, and all modern browsers now reject `SHA-1` certificates.

However, `SHA-1` certificates are still present for legacy purposes, on services that are used by older clients that can not be upgraded. In particular, it remains possible to buy a `SHA-1` certificate today, and there are a few recently-issued certificates in use on the web[7]. There are also a few old `SHA-1` certificates still in use[8]. Those certificates are rejected by modern web browsers, but they can be accepted by non-web TLS clients. For instance, it seems that the Mail application in Windows 10 can open an IMAP session secured with a `SHA-1` certificate without warning. Similarly, OpenSSL still accepts `SHA-1` certificates at security level 1 (the default level in most distributions – but Debian Buster has set the default level to 2, which prevents usage of `SHA-1` certificates).

Chosen-prefix collisions against `MD5` have been able to break the security of certificates in the past, with the creation of a Rogue CA by Stevens *et al.*[27], and in the wild by the flame malware[21]. If some of the CAs still issuing `SHA-1` certificates use predictable serial numbers, a similar attack might be possible today (being located at the beginning of the "to-be-signed" part of the certificate, if the serial number is unpredictable then the CP collision attack is thwarted as a crucial part of the hashed input is not controlled by the attacker).

7.3 `SHA-1` Usage in TLS

Besides certificates, there are two places where `SHA-1` can be used in the TLS protocol: `SHA-1` can be used to sign the handshake, and `HMAC-SHA-1` can be used to authenticate data in the record protocol.

[7] Some examples can be found by searching through certificate transparency logs: `http://web.archive.org/web/20191227165750/https://censys.io/certificates?q=tags%3Atrusted+AND+parsed.signature.signature_algorithm.name%3ASHA1%2A+AND+parsed.validity.start%3A%5B2019-01-01+TO+%2A%5D`

[8] As seen in this scan: `http://web.archive.org/web/20191227165038/https://censys.io/ipv4?q=443.https.tls.validation.browser_trusted%3AYes+AND+443.https.tls.certificate.parsed.signature_algorithm.name%3ASHA1%2A`

Handshake. In order to authenticate the TLS handshake, the client and the server sign a copy of the transcript at the end of the handshake. If the hash function used in the signature is weak, an attacker can use chosen-prefix collisions to mount a man-in-the-middle attack and break various properties of the handshake, as shown by the SLOTH attacks [1]. However, this remains far from being a practical attack, because the CP collision has to be computed in a very short time frame, while the session is being established (timeout value is generally set to a few seconds, but can be up to several minutes).

In TLS 1.0 and 1.1, the handshake is hashed with the concatenation of `SHA-1` and `MD5`. Using the multicollision attack from Joux [9], computing a CP collision for `MD5` ‖ `SHA-1` is not much harder than for `SHA-1`. We give concrete figures in Table 2, showing that this is probably within reach of a well motivated adversary.

In TLS version 1.2, the hash function used is configurable, and is negotiated between the client and the server. `MD5` was one of the possible options, but support has been removed after the SLOTH attack. However, `SHA-1` is still widely supported, and many servers actually *prefer* to use `SHA-1`, even when the client offers better algorithms. Scan results of the top 1M websites show that 3% of them use `SHA-1`[9], and this includes many high profile websites.[10] The vast majority of TLS 1.0/1.1 clients offer `SHA-1` as an option for the signature.

In TLS version 1.3, `MD5` and `SHA-1` have been removed.

Ciphersuites. The ciphersuite used in a TLS connection is the result of a negotiation between the client and server, so it is hard to predict exactly. However, thelarge majority of clients and servers support ciphersuites where `HMAC-SHA-1` is used to authenticate the packets, at least for interoperability reasons. It seems that usage of `HMAC-SHA-1` represents a few percent of all the connections. Telemetry results from Mozilla report about 2% of connections with a `HMAC-SHA-1` ciphersuite.[11] In addition, a scan of websites in the Alexa top 1M show that 8% of them would use a `HMAC-SHA-1` ciphersuite with the client settings used for the scan[12].

This usage is not threatened by our attack, but we recommend to avoid `SHA-1` usage when possible.

OpenSSL. We have first contacted the OpenSSL developers on December 14th.The next version of OpenSSL will no longer allow X.509 certificates signed using `SHA-1` at security level 1 and above (commit `68436f0`). Since security level 1 is the default configuration for TLS/SSL, this will prevent `SHA-1` usage for certificates.

Debian Linux had previously set the default configuration to security level 2 (defined as 112-bit security) in the latest release (Debian Buster); this already prevents dangerous usage of `SHA-1` (for certificates and handshake signature).

7.4 `SHA-1` Usage in SSH

`SHA-1`'s usage in SSH is similar to its usage in TLS. The SSH-2 protocol supports usage of `SHA-1` to sign the transcript (at the end of the key exchange), and `HMAC-SHA-1` to authenticate the data in the record protocol. As in the TLS case, usage of `SHA-1` to sign the transcript has been shown to be potentially vulnerable to the SLOTH attack [1], but

[9] http://web.archive.org/web/20191227174651/https://censys.io/domain/report?field=443.https.tls.signature.hash_algorithm

[10] http://web.archive.org/web/20191227174551/https://censys.io/domain?q=443.https.tls.signature.hash_algorithm%3Asha1

[11] See https://telemetry.mozilla.org/new-pipeline/dist.html#!measure=SSL_CIPHER_SUITE_FULL, were buckets 5, 61 and 63 correspond to `HMAC-SHA-1` ciphersuites

[12] http://web.archive.org/web/20191226134753/https://censys.io/domain/report?field=443.https.tls.cipher_suite.name.raw

this is not practical given the timing constraints (usually just a few seconds, but can be configured to a longer period of time).

Again, the choice of cryptographic algorithms depends on a negotiation between the client and server, so it is hard to know exactly what will be selected. However, scans of the IPv4 space from censys at the time of writing show that roughly 17% of servers use `SHA-1` to sign the transcript[13], and 9% of servers use `HMAC-SHA-1` in the record protocol[14]. This mostly corresponds to servers running old versions of SSH daemons.

OpenSSH. Due to our results, since version 8.2 of OpenSSH a "future deprecation notice" is included, explaining that `SHA-1` signatures will be disabled in the near-future.

7.5 Other Usages of `SHA-1`

DNSSEC. `SHA-1` is still used in DNSSEC, with 18% of the top-level domains using `SHA-1` at the time of writing[15]. Since DNSSEC signatures include user-supplied content, CP collisions could be used to attack the DNSSEC system.

GIT. GIT relies heavily on `SHA-1` to identify all objects in a repository. It does not necessarily require cryptographic security from `SHA-1`, but there are certainly some attack scenarios where attacks on `SHA-1` would matter. In particular, signed GIT commits are essentially signatures of a `SHA-1` hash, so they would be sensitive to collision attacks.

The GIT developers have been working on replacing SHA-1 for a while[16], and they use a collision detection library [26] to mitigate the risks of collision attacks.

Timestamping. Many timestamping servers apparently support `SHA-1`, such as: https://sectigo.com/resources/time-stamping-server

8 Conclusion and Future Works

This work shows once and for all that `SHA-1` should not be used in any security protocol where some kind of collision resistance is to be expected from the hash function. Continued usage of `SHA-1` for certificates or for authentication of handshake messages in TLS or SSH is dangerous, and there is a concrete risk of abuse by a well-motivated adversary. `SHA-1` has been broken regarding collision resistance since 2004, but it is still used in many security systems. We strongly advise users to remove `SHA-1` support to avoid downgrade attacks. We exhibited a practical chosen-prefix collision attack on `SHA-1`, and performed an actual CP collision computation for a reasonable cost. This cost will decrease over time and in a close future will be so cheap that any ill-intentioned person could afford it.

Our work directly impacts the security of PGP/GnuPG Web of Trust: using our CP collision attack on `SHA-1`, we have created two PGP keys with different UserIDs and colliding certificates. This allows an attacker to impersonate any user, as long as trusted third parties sign identity certifications with `SHA-1`.

We also show that gaming or mining GPUs offer a cheap and efficient way to attack symmetric cryptography primitives. In particular, it now costs less than US$ 100k to rent GPUs and break cryptography with a security level of 64 bits (i.e. to compute 2^{64} operations of symmetric cryptography).

[13] http://web.archive.org/web/20191226130952/https://censys.io/ipv4/report?field=22.ssh.v2.selected.kex_algorithm

[14] http://web.archive.org/web/20191226131928/https://censys.io/ipv4/report?field=22.ssh.v2.selected.client_to_server.mac

[15] https://www.dns.cam.ac.uk/news/2020-02-14-sha-mbles.html

[16] https://git-scm.com/docs/hash-function-transition/

Future works. The cost of our attack is roughly four times the cost of a plain collision attack, so there is limited room for improvements in terms of complexity. The factor 4 could be slightly reduced by changing the parameters of the graph, but this is unlikely to result in an improvement of more than a factor 2. Alternatively, improvements to the near-collision search could improve simultaneously identical-prefix and chosen-prefix collision attacks.

On the other hand, we believe there is some possibility to reduce the number of blocks used in the attack without increasing the complexity much. Firstly, with a better use of the global parameters of the general chosen-prefix collision attack. By playing with the number of blocks, the allowable probabilities and the size of the graph, one could probably find a better configuration. Secondly, by not considering only the core differential trail from [23], but using other interesting ones(for example the ones already used in other previous `SHA-1` attacks), we would increase the pool of available differences and in turn reduce the required number of blocks.

A natural future work would be to study other applications of practical chosen-prefix collisions on `SHA-1`. Different protocols would lead to different constraints for the attacker, and unfortunately `SHA-1` remains used on field. What would then be the security impact of our findings on these applications ? What applications could be targeted ?

Finally, it remains to be studied how recent chosen-prefix collision attacks could potentially apply to other hash functions, such as `RIPEMD`, (reduced-round) `SHA-2`, or even others.

We are planning to publish code to make our claims easy to verify, but due to the strong impact of this attack, we will wait until counter-measures are more widely implemented.

Acknowledgements

The authors are grateful to the USENIX Security 2020 reviewers and shepherd for their insightful comments that improved the quality of the paper.

The authors would like to thank Vesselin Velichkov for his help with regards to an initial analysis of neutral bits applicability on `SHA-1` and Werner Koch for his comments on the applicability of our attacks on PGP. The authors would also like to thank `gpuserversrental.com` for their efficient service regarding the GPU cluster renting. The second author is supported by Temasek Laboratories, Singapore.

A small part of the experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`). Development and small scale experiments before launching the main computation were carried out on the rioc cluster from Inria.

References

[1] Karthikeyan Bhargavan and Gaëtan Leurent. Transcript collision attacks: Breaking authentication in TLS, IKE and SSH. In *NDSS 2016*. The Internet Society, February 2016.

[2] Eli Biham and Rafi Chen. Near-collisions of SHA-0. In Franklin [8], pages 290–305.

[3] Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribault, Christophe Lemuet, and William Jalby. Collisions of SHA-0 and reduced SHA-1. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 36–57. Springer, Heidelberg, May 2005.

[4] Gilles Brassard, editor. *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*. Springer, 1990.

[5] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. *RFC 4880 - OpenPGP Message Format*. Internet Activities Board, November 2007.

[6] Ivan Damgård. A Design Principle for Hash Functions. In Brassard [4], pages 416–427.

[7] Christophe De Cannière and Christian Rechberger. Finding SHA-1 characteristics: General results and applications. In Xuejia Lai and Kefei Chen, editors, *ASIACRYPT 2006*, volume 4284 of *LNCS*, pages 1–20. Springer, Heidelberg, December 2006.

[8] Matthew Franklin, editor. *CRYPTO 2004*, volume 3152 of *LNCS*. Springer, Heidelberg, August 2004.

[9] Antoine Joux. Multicollisions in iterated hash functions. Application to cascaded constructions. In Franklin [8], pages 306–316.

[10] Antoine Joux and Thomas Peyrin. Hash functions and the (amplified) boomerang attack. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 244–263. Springer, Heidelberg, August 2007.

[11] Vlastimil Klima. Tunnels in hash functions: MD5 collisions within a minute. Cryptology ePrint Archive, Report 2006/105, 2006. http://eprint.iacr.org/2006/105.

[12] Gaëtan Leurent and Thomas Peyrin. From collisions to chosen-prefix collisions application to full SHA-1. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 527–555. Springer, Heidelberg, May 2019.

[13] Gaëtan Leurent and Thomas Peyrin. SHA-1 is a Shambles - First Chosen-Prefix Collision on SHA-1 and Application to the PGP Web of Trust. Cryptology ePrint Archive, Report 2020/014, 2020. https://eprint.iacr.org/2020/014.

[14] Marc Stevens. Attacks on Hash Functions and Applications. PHD Thesis, Leiden University, June 2012.

[15] Ralph C. Merkle. One Way Hash Functions and DES. In Brassard [4], pages 428–446.

[16] National Institute of Standards and Technology. FIPS 180-1: Secure Hash Standard, April 1995.

[17] National Institute of Standards and Technology. FIPS 180-2: Secure Hash Standard, August 2002.

[18] National Institute of Standards and Technology. FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, August 2015.

[19] Ronald L. Rivest. The MD4 message digest algorithm. In Alfred J. Menezes and Scott A. Vanstone, editors, *CRYPTO'90*, volume 537 of *LNCS*, pages 303–311. Springer, Heidelberg, August 1991.

[20] Ronald L. Rivest. *RFC 1321: The MD5 Message-Digest Algorithm*. Internet Activities Board, April 1992.

[21] Marc Stevens. Counter-cryptanalysis. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 129–146. Springer, Heidelberg, August 2013.

[22] Marc Stevens. New collision attacks on SHA-1 based on optimal joint local-collision analysis. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 245–261. Springer, Heidelberg, May 2013.

[23] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 570–596. Springer, Heidelberg, August 2017.

[24] Marc Stevens, Pierre Karpman, and Thomas Peyrin. Freestart collision for full SHA-1. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 459–483. Springer, Heidelberg, May 2016.

[25] Marc Stevens, Arjen K. Lenstra, and Benne de Weger. Chosen-prefix collisions for MD5 and colliding X.509 certificates for different identities. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 1–22. Springer, Heidelberg, May 2007.

[26] Marc Stevens and Daniel Shumow. Speeding up detection of SHA-1 collision attacks using unavoidable attack conditions. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017*, pages 881–897. USENIX Association, August 2017.

[27] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen K. Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 55–69. Springer, Heidelberg, August 2009.

[28] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, January 1999.

[29] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 17–36. Springer, Heidelberg, August 2005.

A Improving SHA-1 Collision Attack

We now give the technical details of the improvements introduced in Section 3, for readers that are familiar with previous works [22, 23, 12].

A.1 Boomerangs and Neutral Bits

As explained in [12], an important factor to evaluate the cost of the attack is the number of boomerangs available when looking for a conforming message. The collision attack from Eurocrypt 2013 [22] and its GPU implementation from Crypto 2017 [23] (the "Shattered" attack) use three boomerangs, on bits 6 and 8 of M_6 (red type in Figure 9), and on bit 7 of M_9 (blue type in Figure 9). However, the boomerang on $M_6[8]$ actually flips the value of $W_{77}[0]$ due to the message expansion, and breaks the condition $W_{77}[0] \oplus W_{77}[2] = 1$ listed on Table 5 of [23]. This reduces the probability of the trail on rounds 61—80 by a factor $3/4$, from $2^{-19.17}$ to $2^{-19.58}$. However, the boomerang almost doubles the number of partial solution produced, so that using it still improves the attack.

In addition, we realized that the "Shattered" attack [23] uses a neutral bit on $M_{13}[11]$. However, this breaks the condition $W_{76}[0] \oplus W_{76}[1] = 1$, and reduces the probability of the trail by a factor roughly $2^{0.2}$. In our analysis, we assume that this neutral bit has been removed: since it has a very small effect on the number of partial solution produced, it reduces the complexity of the attack by a factor $2^{0.2}$. In particular, this is why we consider that a collision requires $2^{48.5}$ A_{33}-solutions rather than $2^{48.7}$ (where an A_i-solution refers to an input pair that is following the differential path until word A_i inclusive).

In our chosen-prefix attack, we also need conditions on $W_{77}[0]$, and we decided to remove the boomerang on $M_6[8]$ in order to keep more control on the output difference and to simplify the attack. In particular, removing this boomerang makes easier the construction of trails with the extra constraints (and we have successfully built trails with the two remaining boomerangs for all successive blocks). This implies that the cost of near-collision blocks increases by a factor $3/2$ compared to the "Shattered" attack, leading to $C_{\text{block}} = 2^{64.9}$ on a GTX 970 (after gaining a factor $2^{0.2}$ by removing $M_{13}[11]$ as a neutral bit). Therefore we can estimate more accurately the complexity of the previous attack [12] as $2^{67.1}$ SHA-1 computations, instead of the range of $2^{66.9}$ to $2^{69.4}$ reported previously (the optimal attack parameter choice for [12] is then a maximum cost of 3.5 C_{block}).

As an optimization, we decided to use the boomerang on $M_6[8]$ for the last block, because the computation of the last block is identical to the second block of an identical-prefix collision attack. In particular, we had no trouble building a path with this boomerang, and this results in a speed-up of a factor 1.9 in the rate of A_{33}-solutions. We need to increase the number of solutions by a factor $4/3$ because the solutions are of lower quality, but this still corresponds to a speed-up factor of roughly 1.4. Since the last block represents a significant part of the total computation, this is a worthwhile optimization.

A.2 Additional Boomerangs and Modular Correction

We found out that in addition to previously mentioned boomerangs, we can use very short boomerangs on bits 4, 5, and 6 of M_{11} (green type in Figure 9), with a single correction on M_{12}. These boomerangs are neutral until step 22, like the boomerang on M_9. The problem is that they will clash with existing small boomerangs starting at $M_6[6]$, $M_6[8]$ and $M_9[7]$.

More precisely, the boomerang starting at $M_{11}[4]$ will flip the message condition "$\overline{\texttt{x}}$" from the boomerang starting at $M_6[6]$, corresponding to the last message correction of the local collision. In order to avoid this issue, we simply change the last correction of the $M_6[6]$ boomerang to be a modular addition correction instead of an XOR correction. This

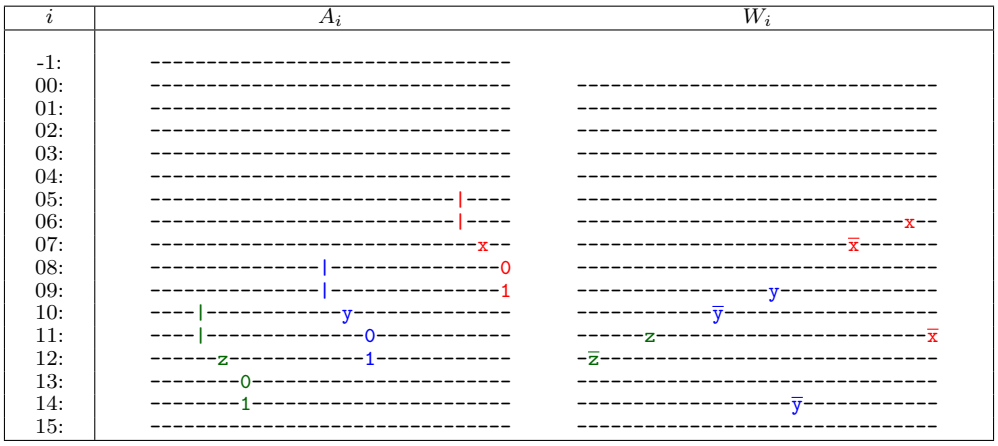


Figure 9: Boomerangs' differential paths used for `SHA-1` with the corresponding constraints forced in order to have probability one in the 16 first steps. The red one (perturbation x) represents a small boomerang (named AP_1 in [**?**]) composed of a single local collision starting on M_6, here positioned at bit $j = 2$. The blue one (perturbation y) represents another small boomerang used in [22, 23], also composed of a single local collision, but starting on M_9, here positioned at bit $j = 14$. The green one (perturbation z) represents a new and even smaller boomerang built from a partial local collision starting on M_{11}, here positioned at bit $j = 25$. The MSB's are on the right and "`-`" stands for no constraint. The letters represent a bit value and its complement is denoted by an upper bar on the corresponding letter. The notation "`|`" on two bits vertically adjacent mean that these two bits must be equal.

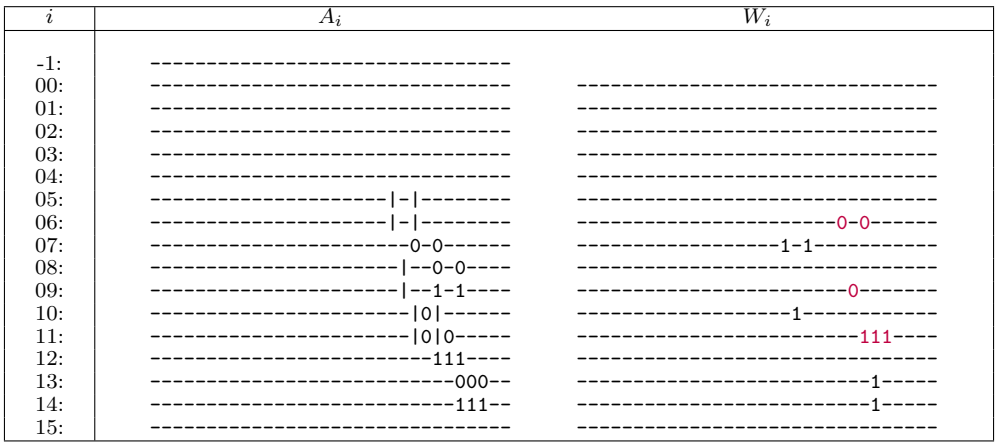


Figure 10: Exact conditions required to prepare all the boomerangs' differential paths used for our CP collision attack on `SHA-1` with the corresponding constraints forced in order to have probability one in the 16 first steps. The MSB's are on the right and "`-`" stands for no constraint. The notation "`|`" on two bits vertically adjacent mean that these two bits must be equal. These conditions basically correspond to very short boomerangs started at $M_{11}[4]$, $M_{11}[5]$ and $M_{11}[6]$, and small boomerangs started at $M_6[6]$, $M_6[8]$ and $M_9[7]$ (boomerang starting perturbations are marked in purple). We remark an extra condition $M_{13}[5] = 1$ in order to potentially correct the clash between boomerangs $M_{11}[5]$ and $M_9[7]$.

will naturally correct the perturbation as the step operation involved is indeed a modular addition and the boomerang on $M_6[6]$ will behave as expected (the condition "$\overline{\texttt{x}}$" is actually not needed anymore). This can be seen as a generalization of the boomerang strategy used so far for `SHA-1`: boomerang corrections can be applied modular addition-wise instead of XOR-wise. This will induce changes in subsequent bits in the corresponding message words because of carry propagations that might naturally occur with a modular addition operation, but as long as these bit changes do not mess with existing message conditions, we are fine[17]. This idea might also be interesting to analyse other hash functions.

Similarly, the boomerang starting at $M_{11}[6]$ will flip the message condition "$\overline{\texttt{x}}$" from the boomerang starting at $M_6[8]$, corresponding to the last message correction of the local collision. This is exactly the same situation and we avoid this issue by changing the last correction of the $M_6[8]$ boomerang to be a modular addition correction instead of an XOR correction.

Finally, the boomerang starting at $M_{11}[5]$ will flip the condition "`1`" in the internal state, required for the boomerang starting at $M_9[7]$. This will create an uncontrolled difference when we use the $M_9[7]$ boomerang, that we correct by introducing a new difference in $M_{13}[5]$ (setting the extra condition $M_{13}[5] = M_9[7] \oplus 1$ will ensure a proper correction using XOR correction). Note that we perform this further correction only in the case where the boomerang starting at $M_{11}[5]$ was triggered. We observe furthermore that the correction on $M_{13}[5]$ will maintain a good quality of neutrality for $M_{11}[5]$ boomerang. All the conditions required to use the boomerangs are given in Figure 10.

[17] In more details, since the boomerangs on $M_{11}[4]$, $M_{11}[5]$ and $M_{11}[6]$ are very short and thus applied before boomerangs $M_6[6]$, $M_6[8]$ and $M_9[7]$, we don't actually care if the last ones break conditions of the first ones since they have already been used.

B Chosen-prefix Collision Example

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQQNBH/oF4ABIAD/S2V5IGlzIHBhcnQgb2YgYSBjb2xsaXNpb24hIEl0J3MgYSB0
cmFwIXnGGvCvzAVFFdknTnMHYksdx/sjmIu43otXXbp7nqsxwWdLbZdDeKgncy/1
hRx2ouYHcrWkfOHqxAu5k8EtjHDiSk+NX83twbMsnPGeMa8kKXWdQuTf2zFxn1h2
I+5VKTm23NxFn8pTVTtw+H7eMKJH6jr2x1mi8gsyDXYNtk/OeQhP08yzzdSDYtlq
nEMGF8r/bDbGN+U/3ihBf2Jv7FTteUOkbl9XMPK7OPsd9uAJABDQDiSteL+SZBmT
YI6NFYp4nzTEb+HmAn81pMv7gnB2xQ7KDot8ymm7LCt5Aln5v5Vw3Y1EN6MRX6/3
w8rAmtJSZgVcJxBHVReOrv+CWiyqKs+13mTOdkHcWaVBqfycdWdW4uI9xxPIwkyX
kKprDjin9V8URSocooUN3ZVi/ZoYrUJJaqlwCPdGcvaO9GHriLCZM9YmtPkYdJzA
J/3dbEJfxCFoNdATTRUoW6sst4Sk98u0+1FNS/D2I3zwCp6fEyuaBm5v0X9sQph0
eFhv9lGvlnR/tCa5hyuaiOQGP1m7M0zABlD4OoDEJ1G3GXTTAPwoGaLo8eMsG1HL
GOa/xNubrvZ11Kr1sVdKBH+PbdLsFTqTQSKTlO2Sj4jO2TY8/vl84udCvzTJa47z
h1Z2/qXMqOX33qC6skE9TeAO5x7gHxYr22Oer9kl5q66rmo1TvF88gWkBPvbEvxF
TUH92VzyRZZkoqODLR2mCnMmQHXX8eDWwUA656DYYd8/5XBxiN1eB9FYm5+LZjBV
P4/DUrPgwn2oC926TGQCDQYH9nQNLoASAgwRjxlZlJl+xnAClehPhvimlPrsylEs
iZXJZuZjfmF1R1VUbBDRXshFngA85nlvVon7fAnZTYNiyM5PM5ZBHTH/91uEWef+
vMgBV5t3S1TZnD6dQpgStXi3kNZ77juY93ADOo64i5vmC8TONoSzg7AO+6KzQ21R
+zBO/AKu92MeHNcAEQEAAdEAAAEZwFcBEAABAQAAAAAAAAAAAAAAAP/Y/9sAQwD/
////////////////////////////////////////////////////////////////
////////////////////8AACwgAQABYAQERAP/EACgAAQEBAAAAAAAAAAAAAAAA
AAAEAxABAAAAAAAAAAAAAAAAAAAAAP/aAAgBAQAAPwDQTqABOoAE6gE6gATgAKAT
ihOChOKE4AAAKE4AChOKE6gATqE6gE4oTigH/9ldulZvOS5WXbOQhggKNiGWIm/z
IV/tc5s7KOrrL9gZCVAvnv1D5pEAuEqDOBDdABEBAAGOGUFsaWNlIDxhbGljZUBl
eGFtcGxlLmNvbT6JBFQEEwEIAD4WIQRDzVxbBP9XQvoUGryp11WpY1SMeAUCf+gX
gAIbLwUJAAFRgAULCQgHAgYVCgkICwIEFgIDAQIeAQIXgAAKCRCp11WpY1SMeJHD
H/9jJj4FARmtNhFXqDhNcNV6C3mQAevQR1iCbFwWHXdd2yRfyFv9/dVY4Wa2Q16j
BuCUhl7SDmO1IYlYBBo+91Kr11rDezL/y/PQQsKzRp9nEoOr+QCVnIozjJ1DeP77
Xwxu77gRe1QkIqlOaVUhP6vPRD+oFgViYUUWIdNgh4bBJGDMY43BJyeCxmuqRo8/
sHTWu7rmhuHpsQiTRd5valBPHXbZKAcS+TmgGM8dMv3NOldq1wDZWDrWfD6vCmnP
a4hH3UybM9G6kkkQ2xETt7hyW6XlsLXk9CtaZ2hz1kUUktTa8LU1mj9T8dsLbTjO
qipYps/ry0Cro7gkaYiNGeeelNzPhfz8jXHrK47hZOiZbcb7JUcR6WodGAWV6bFE
+Jj02NOnX87QsWk1z5YdHsvDEbzbE9fQZ2r8hPEDsqEDXsSSw6h/FkhCcxwn9RV2
SGt5kZkUOqGP6chAITiGEEX4OTcnfgyNvSJcHJ2iMaKGEaaQv5MfaefMRX9tGiz3
wmTYxSdUqkFiG2YtOkZWdCZDBKVL6Y8g7fM6G5DQc+/4alVqGdZi1TwAmKcZ142E
2JL6MAu9zfmjMcRkXytzUfkKGKa/kpAMPsn1cWYxlQe0PxxVAM8iZTPUSnBah13t
Ec1fx9jJ1cJkHBW7w1AFS+kLKDOblX9rreTUDzBQLPkaXWqH5fLZvc+EGJpt6roL
obVZ2mVDfmYNABYnw4nTDBGb2Vk6OMpPhOk7pMMbXJA9kuwvgYa6VztyIZcyi8Mz
LOKbtwsogOS5KVO1tOcKMGtXpni6Vsx/k1ydvtOBUuG7VjG8Y1EuGyOUzdIuXe1r
UUTIV7GZfNpzxAumvoY2umKJQzpA/jG1yban05IViIIFotKCfA8cpKzY1roUtyjq
FPgBOT2YUadFStarrypb+f3SnjOnK4UqyJOYbJQiKbo/NVvY2cQdqR5wyBVKV7RR
qjkNArxkHuORUaBnywzywztOHNsOojX3IyOeytLcIrbaKYJYaeZCwOWcNIWP5avq
u4Lkfq+ariSGdbfHKBR1OJWBeD/Rne+E9rmwrvk2S8fd1pdx44R8RzIvWfCKUQeu
wnfHBsP5BOZkjltSQnpBHsNSFnie0o5uSKnL8dyTbH3TtX1u3D7S95KZRley+n+5
hfQTTHWo8Xq/m6wrOTYxdA4DbaSJJ/wZFHOYaybXXkXhIQY3n3o+4mdgBQy3okIG
enWdH6Jdty6xrDimgxMjOUIyQ0JTlHBksIIViqzaY52ATI2U5UBpK7DgNC+IzIFe
QVWNIFVqT3Ifqcfd5iKZZvEEKt4QWDfIvc5bPkXraWV8Aqq3ylOiYiBFRVLVDBy3
AXrthG5XMjPFw/7TQEoIY5MYiQEcBBABAgAGBQJ/6VrwAAoJEK+7H+1pUalWFnOH
/AmChC2A4mXbw6m6hIs/zWUTNRAM4I9daUst138jmNFKUz67zIxUvUeUuU2n6Lfj
rerpVX8LuQbz2/e+g1AIcAKBRniBDyjzedWuya/faLW9rSpATUy/3Rv2GOT1QBSw
fsQN1NT8oBbMYZl7ht0ylst2m9zJyI+XjktMiIIxfr8zlMjIpXU9xF0YPuSNMOSm
l38jeLHN9hHZpvsvsK2EmhCTF2KhHvozlFfJ8wr1OcnWxdEzoYntx178TyOKRgC+
oTLIccOSrNKjzkRuNnOsEEpAudFR6yxx8HLZatSXWpzitNdaG8APDwJiS5AHaqVQ
yNnDiF8Aoz5caQEw4D8aDvk=
=O4wQ
-----END PGP PUBLIC KEY BLOCK-----
```

Figure 11: Key A

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQMNBH/oF4ABGAD/UHJhY3RpY2FsIFNIQSOxIGNob3Nlbi1wcmVmaXggY29sbGlz
aW9uIRObGumYeEEDh99dn8HYkndx/szLIu4wrdXXb7Hnqsr4WdLfbNDeLTLcy/h
iRx2oCYHcqUQfOH26Au5l30tjGhSSk+dX83tzQssnOGSMa8m6XWdU1Df2y1Nn1hy
n+5VMxm23Mxhn8pPuTtw7HLeMKCH6jrmc1mi7icyDXKxtk/syQhPw8yzzdg7Ytl6
kEMGFQr/bCZyN+Uj4ihBe95v7E7NeUOOS19XLB67OO8R9uALwBDQHpCteKO+ZBmX
3I6NDTp4nyTEb+Hqun81tMf7gnK2xQ7auot81lW7LC/FAlnjn5VwzalEN7/9X6/j
z8rAmBJSZhXoJxBbeReOqkOCWjQaKs+l3mTOevncWbVNqfyetWdW8lY9xw/0wkyT
LKprFBin9U8wRSoAToUNyZli/ZjYrUJZ3qlwFNtGcvIy9GHzOLCZI9YmtPWgdJzQ
K/3dboJfxDHcNdAPcRUoXxcst56E98uk31FNVxz2I2j8Cp6d0yuaFtpvOWNAQphw
xFhv7uGvlmR/tCa1PyuamOgGP1t7MOzQslD4JrzEJ1ULGXTJIPwoCYbo8f/AG1Hf
FOa/xhubrubB1KrpnVdKAMOPbcpcFTqDQSKTm/WSj5jC2TY+Pv1881NCvyj1a473
O1Z25IXMqPXT3qCmXkE9Wew05xwgHxY7b2Oes/Ul5qoGrmot/vF84gWkBPdjEvxV
QUH925zyRYbQoqOfER2mDs8mQG/38eDG5UA6+0zYYcsz5XBzSN1e2F2Ym4OnZjBR
g4/DSgPgwm2oC9229GQCHQYH9nQNLoASAgwRjxlZlJl+xnAClehPhvimlPrsylEs
iZXJZuZjfmF1R1VUbBDRXshFngA85nlvVon7fAnZTYNiyM5PM5ZBHTH/91uEWef+
vMgBV5t3S1TZnD6dQpgStXi3kNZ77juY93ADOo64i5vmC8TONoSzg7AO+6KzQ21R
+zBO/AKu92MeHNcAEQEAAbQVQm9iIDxib2JAZXhhbXBsZS5jb20+iQNUBBMBCAA+
FiEExr/i/LvlGokr63eYEjPUzGHb2cQFAn/oF4ACGy8FCQABUYAFCwkIBwIGFQoJ
CAsCBBYCAwECHgECF4AACgkQEjPUzGHb2cSLZhgAynF426xC9qX3Afh2jzWUFWMZ
duiZ78JwpopGRo/BnAzlD/dmmK/OjaVfG7aCq/R/Fd3faeA1qxJuST6Z2bwDCbtj
HjERNPATekCJaHP4EgHsFSDZQUNdUWswO7rclXnHmN7AxCkfXg+U5R1cxQkyBIL7
U9uxleMU6+Fr+QP2y4MsZpM1n1ipWJK1FuKKQeNY9J7fRdr0LyqiVfYOTqtI6kr2
tzk1oQ2CsiRmidCOMmAaW1L5tppmDiPMo3Yqv0lCrqeyfnOBpwOLFIz9DH6Ndf1z
213eDWgUpHbHvERSOrwxv/4eizUGmisxgjrWwjhdRch7ZNWQZYQVwtcjz7wWhVPW
vaS+t86YayrDO8J1jw7FfkFdPHlt1TlawWcGsiehiB9EazJFM4zGw/IYjQWLhBvv
e+NeKrSjIso3ZXSbqVdl/Dgj4Go9J3KXXqJSbaxedak3SX44XwO54NAgYsjlOak5
yHKdBjrZYdtElzkkJklUEYdQyA5icRjJ2Kiy3g4xlR+7ev6i1QSoouBIbQD7qmeO
45iKG17mOe6qMPrPB9sTUue/gVfZaysR4nUXIzIIk6sHg2Q7Fl1ohYO5SFYvFVrj
dtpGO1XeKXnEKaUb9Z9OCUlh16SRL7/xXq86SKqi7SoIMNFspUmsuk4np/e6ipBn
u2cAjpThNrELZjmvh0KCt/x6jv1vbdUiB+gg2SbTD5qFmoir2OXfKlNrz6PUhGI4
dQFKbBsmbJtUqTDEogRlHUZ2CXWpj5Zs9xvcGvaTpurqncoEmz96SKyhiPSpDw0x
tIxEaupvGnug4/CtlvFE+vP7ZzucMSThLLXJVRATNWcCWosfXXWv4A7Lebbxqj2P
ukwJFmmJOet8s2h6blFRm3MegHP7hqjqQe5XovZLqwLKrr3DqdZYhmxOjeQATo1I
BKyviJnh8XU293hHv8M4wGRVPvyMrTfwcMj+pgb4yTPShoysONWIE954c3yic94+
slaK45askyHC9ORw8oraPOyEOI9ewSfi5N3zwXImiQEcBBABAgAGBQJ/6VrwAAoJ
EK+7H+1pUalWI6UH/3HgRCkMiyDRrRi8lZrMWcf0xIlC1kiSb62Kt2VhsG/RL2tO
GJZhrOp3+fg3QiRFMnE+5ScJIBhL3PgScTEOoCcmA3+jo7s8fnfhLVzbHjcJEtxP
fo4zlr9pKh5cUXlsSrZPXCHE/1IdEXjbETie6fS9aSWhnDYnBdwTw8CrHRY/tibn
tMla3m8w89zfTcDVkHNSgj6nSQly+jlU+3I5Ny+pJGxEytMeBwyWoCOnR/qltm7U
kG2yMV83tqkSVqMyIPGx5MQuc21sDWTmP3ct+VMrJD448+S2Emor0InhmkRuT5bS
7p8AHuEkhlPITSRvGpgQjnzDGuP6TUbVUGReTMvRwFnAVwEQAAEBAAAAAAAAAAAA
AAAA/9j/2wBDAP//////////////////////////////////////////////////
//////////////////////////////wAALCABAAFgBAREA/8QAKAABAQEA
AAAAAAAAAAAAAAAAAQDEAEAAAAAAAAAAAAAAAAAAAAAA/9oACAEBAAA/ANBOoAE6
gATqATqABOAAoBOKE4KE4oTgAAAoTgAKE4oTqABOoTqATihOKAf/2V24tm85L1Zd
vRCGCAo2IZYib/MhX+1zmzso6usv2BkJUC+e/UPmkQC4SoM4ENOAEQEAAbQAAAAZ
QWxpY2UgPGFsaWNlQGV4YW1wbGUuY29tPokDVAQTAQgAPhYhBMa/4vy75RqJK+t3
mBIz1Mxh29nEBQJ/6BeAAhsvBQkAAVGABQsJCAcCBhUKCQgLAgQWAgMBAh4BAheA
AAoJEBIz1Mxh29nEsnkYALms73X14/+0sUPjojsWYeUzFSY67/ihObZBEMcp3qoz
4E5kFlg2ZquAEBnMa3IANy7kvZprxgYd3EABzaBUFkMsrGovOaaJCLmtSvWP9ooh
0vWgHIOtuTUZGBcdJhNORnZ/BOW1azlrYGdzIiQOzMn+67BfUkJIIoHgdNOOZZkE
f2fncC3vmyh9/8QhsUF553mOxo8VYYNSO/S/GSr6MD1eP+YpA3v2DznaHyTzwFmN
l4apcEErudob/PSzn9SdHDjwC2WxqObEWllLVtnANi1GP+sSTVbWCREJ+m72Vljv
ADP5nKD3HPKeqoRNuQkv2TdQdkyTp3UBechnqK+u7xl72dOZGJ9EOnT/o23zbxFz
vrIcBbCQnqrtZnwUDIaxG8numABkfM6T5mOzOetIOctQoRw7MQsB4C6VxqDtho3k
5dSomtsiNsiuLlI6FT5YIUJWOV7r8UVzrAKq98buaAHlnBhpps2WHBrm8oOBO16g
rJOEfhgYyB6cMG6OhLAhkjPVbu5HxBngylu1ZrotktK+eWGtWyZRA6/pZN5t7EdO
IzOiS6Ewp5yBHB5t24ZytrZhVPgExOka47yEJMKq8Q+uM8V3kfsk6+Kb6aK2cJyg
A1igSL9T48a9es0dWm+rPxKg2Sb1+QdWCWpG52Nb4d+EfqQERdIgp97dgJaZ3ye7
ORR+tK9gpoJHDI1UBdm3M8PeYhiMbOWWvuIsuB5+/fstLTNngmyT+gqwr4F3XmG/
3Zoc1gmXxRG7VYn3DiT7PLZJmSovpuGay2pqAQvc+nJrjCOw67qUmBm8QHwbMWMh
xZkfBLfD6V/7iWCIhPDmJteKW/pxBASJh9gA9h8T3BRTTOSM252aaG6EYF5Wc+ha
/rhl1c6xSE6SEArf+aYcrz4KXap4HDC4pYqJIltWg8uvs2+qejYH4iW3KArmmfjU
3naa7LDUKEs7U2tcPyg1ZfTC529sROq5OgJ1agcaHr8RJdIZ6qGVdaBQLMG9utUW
6OuHZmhCPy+lS3Ftjp19/YkBHAQQAQIABgUCf+la8AAKCRCvux/taVGpVhZ9B/wJ
goQtgOJ128OpuoSLP81lEzUQDOCPXW1LLdd/I5jRSlM+u8yMVL1HlLlNp+i3463q
6VV/C7kG89v3voNQCHACgUZ4gQ8o83nVrsmv32i1vaOqQE1Mv90b9htE9UAUsH7E
DZTU/KAWzGGZe4bdMpbLdpvcyciP145LTIiCMX6/M5TiyKV1PcRdGD7kjTNEppd/
I3ixzfYR2ab7L7CthJoQkxdioR76M5RXyfMK9dHJisXRM6GJ7cde/E8tCkYAvqEy
yHHDkqzSo85EbjZzrBBKQLnRUesscfBy2WrU1iqc4rTXWhvADw8CYkuQB2qlUMjZ
w4hfAKM+XGkBMOA/Gg75
=T/Dd
-----END PGP PUBLIC KEY BLOCK-----
```

Figure 12: Key B

	Message A	Message B
0x0000	99 04 0d 04 7f e8 17 80 01 20 00 ff 4b 65 79 20 69 73 20 70 61 72 74 20 6f 66 20 61 20 63 6f 6c 6c 69 73 69 6f 6e 21 20 49 74 27 73 20 61 20 74 72 61 70 21 79 c6 1a f0 af cc 05 45 15 d9 27 4e	99 03 0d 04 7f e8 17 80 01 18 00 ff 50 72 61 63 74 69 63 61 6c 20 53 48 41 2d 31 20 63 68 6f 73 65 6e 2d 70 72 65 66 69 78 20 63 6f 6c 6c 69 73 69 6f 6e 21 1d 27 6c 6b a6 61 e1 04 0e 1f 7d 76
0x0040	73 07 62 4b 1d c7 fb 23 98 8b b8 de 8b 57 5d ba 7b 9e ab 31 c1 67 4b 6d 97 43 78 a8 27 73 2f f5 85 1c 76 a2 e6 07 72 b5 a4 7c e1 ea c4 0b b9 93 c1 2d 8c 70 e2 4a 4f 8d 5f cd ed c1 b3 2c 9c f1	7f 07 62 49 dd c7 fb 33 2c 8b b8 c2 b7 57 5d be c7 9e ab 2b e1 67 4b 7d b3 43 78 b4 cb 73 2f e1 89 1c 76 a0 26 07 72 a5 10 7c e1 f6 e8 0b b9 97 7d 2d 8c 68 52 4a 4f 9d 5f cd ed cd 0b 2c 9c e1
0x0080	9e 31 af 24 29 75 9d 42 e4 df db 31 71 9f 58 76 23 ee 55 29 39 b6 dc dc 45 9f ca 53 55 3b 70 f8 7e de 30 a2 47 ea 3a f6 c7 59 a2 f2 0b 32 0d 76 0d b6 4f f4 79 08 4f d3 cc b3 cd d4 83 62 d9 6a	92 31 af 26 e9 75 9d 52 50 df db 2d 4d 9f 58 72 9f ee 55 33 19 b6 dc cc 61 9f ca 4f b9 3b 70 ec 72 de 30 a0 87 ea 3a e6 73 59 a2 ee 27 32 0d 72 b1 b6 4f ec c9 08 4f c3 cc b3 cd d8 3b 62 d9 7a
0x00c0	9c 43 06 17 ca ff 6c 36 c6 37 e5 3f de 28 41 7f 62 6f ec 54 ed 79 43 a4 6e 5f 57 30 f2 bb 38 fb 1d f6 e0 09 00 10 d0 0e 24 ad 78 bf 92 64 19 93 60 8e 8d 15 8a 78 9f 34 c4 6f e1 e6 02 7f 35 a4	90 43 06 15 0a ff 6c 26 72 37 e5 23 e2 28 41 7b de 6f ec 4e cd 79 43 b4 4a 5f 57 2c 1e bb 38 ef 11 f6 e0 0b c0 10 d0 1e 90 ad 78 a3 be 64 19 97 dc 8e 8d 0d 3a 78 9f 24 c4 6f e1 ea ba 7f 35 b4
0x0100	cb fb 82 70 76 c5 0e ca 0e 8b 7c ca 69 bb 2c 2b 79 02 59 f9 bf 95 70 dd 8d 44 37 a3 11 5f af f7 c3 ca c0 9a d2 52 66 05 5c 27 10 47 55 17 8e ae ff 82 5a 2c aa 2a cf b5 de 64 ce 76 41 dc 59 a5	c7 fb 82 72 b6 c5 0e da ba 8b 7c d6 55 bb 2c 2f c5 02 59 e3 9f 95 70 cd a9 44 37 bf fd 5f af e3 cf ca c0 98 12 52 66 15 e8 27 10 5b 79 17 8e aa 43 82 5a 34 1a 2a cf a5 de 64 ce 7a f9 dc 59 b5
0x0140	41 a9 fc 9c 75 67 56 e2 e2 3d c7 13 c8 c2 4c 97 90 aa 6b 0e 38 a7 f5 5f 14 45 2a 1c a2 85 0d dd 95 62 fd 9a 18 ad 42 49 6a a9 70 08 f7 46 72 f6 8e f4 61 eb 88 b0 99 33 d6 26 b4 f9 18 74 9c c0	4d a9 fc 9e b5 67 56 f2 56 3d c7 0f f4 c2 4c 93 2c aa 6b 14 18 a7 f5 4f 30 45 2a 00 4e 85 0d c9 99 62 fd 98 d8 ad 42 59 de a9 70 14 db 46 72 f2 32 f4 61 f3 38 b0 99 23 d6 26 b4 f5 a0 74 9c d0
0x0180	27 fd dd 6c 42 5f c4 21 68 35 d0 13 4d 15 28 5b ab 2c b7 84 a4 f7 cb b4 fb 51 4d 4b f0 f6 23 7c f0 0a 9e 9f 13 2b 9a 06 6e 6f d1 7f 6c 42 98 74 78 58 6f f6 51 af 96 74 7f b4 26 b9 87 2b 9a 88	2b fd dd 6e 82 5f c4 31 dc 35 d0 0f 71 15 28 5f 17 2c b7 9e 84 f7 cb a4 df 51 4d 57 1c f6 23 68 fc 0a 9e 9d d3 2b 9a 16 da 6f d1 63 40 42 98 70 c4 58 6f ee e1 af 96 64 7f b4 26 b5 3f 2b 9a 98
0x01c0	e4 06 3f 59 bb 33 4c c0 06 50 f8 3a 80 c4 27 51 b7 19 74 d3 00 fc 28 19 a2 e8 f1 e3 2c 1b 51 cb 18 e6 bf c4 db 9b ae f6 75 d4 aa f5 b1 57 4a 04 7f 8f 6d d2 ec 15 3a 93 41 22 93 97 4d 92 8f 88	e8 06 3f 5b 7b 33 4c d0 b2 50 f8 26 bc c4 27 55 0b 19 74 c9 20 fc 28 09 86 e8 f1 ff c0 1b 51 df 14 e6 bf c6 1b 9b ae e6 c1 d4 aa e9 9d 57 4a 00 c3 8f 6d ca 5c 15 3a 83 41 22 93 9b f5 92 8f 98
0x0200	ce d9 36 3c fe f9 7c e2 e7 42 bf 34 c9 6b 8e f3 87 56 76 fe a5 cc a8 e5 f7 de a0 ba b2 41 3d 4d e0 0e e7 1e e0 1f 16 2b db 6d 1e af d9 25 e6 ae ba ae 6a 35 4e f1 7c f2 05 a4 04 fb db 12 fc 45	c2 d9 36 3e 3e f9 7c f2 53 42 bf 28 f5 6b 8e f7 3b 56 76 e4 85 cc a8 f5 d3 de a0 a6 5e 41 3d 59 ec 0e e7 1c 20 1f 16 3b 6f 6d 1e b3 f5 25 e6 aa 06 ae 6a 2d fe f1 7c e2 05 a4 04 f7 63 12 fc 55
0x0240	4d 41 fd d9 5c f2 45 96 64 a2 ad 03 2d 1d a6 0a 73 26 40 75 d7 f1 e0 d6 c1 40 3a e7 a0 d8 61 df 3f e5 70 71 88 dd 5e 07 d1 58 9b 9f 8b 66 30 55 3f 8f c3 52 b3 e0 c2 7d a8 0b dd ba 4c 64 02 0d	41 41 fd db 9c f2 45 86 d0 a2 ad 1f 11 1d a6 0e cf 26 40 6f f7 f1 e0 c6 e5 40 3a fb 4c d8 61 cb 33 e5 70 73 48 dd 5e 17 65 58 9b 83 a7 66 30 51 83 8f c3 4a 03 e0 c2 6d a8 0b dd b6 f4 64 02 1d

Figure 13: Chosen-prefix collision for `SHA-1`. The colors show the prefix, the birthday bits, and the near-collision blocks. Both messages lead to the same `SHA-1` value: `8ac60ba76f1999a1ab70223f225aefdc78d4ddc0`

Table 9: Differences in the state after each block, and output differences at the end of the trail (before the feed-forward).

Block	State difference
Birthday	----------n---n---u-n-u-n-u-n-u-- u----------n---n-u-n-u-n-u-- ----------n-----n----- ----------n-u-n-u-n-u-n-u-n
Block 1	u----------n-u-n---u-n-u-n-u-- ----------n-u-n-u-n-u-n-n-- ----------n-u---- u----------n-u-
Block 2	----------n-u-------n-u-- ----------n-u-n-u-n-u-- ----------u ----------n-u-
Block 3	----------n-----n-n-n-n-n-n-n- ----------n-n-n-n----- ----------n----- ----------
Block 4	----------n-u-n-u-n-u-n-u-n-u- ----------n-u-n-u-n-u- u----------n-u----u ----------
Block 5	----------n-----n-u-u-----u-n-n- ----------n-----n-n-n-n- u----------n---- ----------
Block 6	----------n-n-------n-n-n--- ----------n-n-n---n--- u---------- ----------n-n-u-
Block 7	----------n---n-n-n-n-u--- ----------n---n-n-n--- ---------- ----------n--- ----------
Block 8	----------n---n-n-n-n-u- ----------n---n-n-n-u- ---------- ----------n----- ----------
Block 9	---------- ---------- ---------- ---------- ----------

Block	Output difference
Block 1	u----------u-n-u-n-u-n-u-n-u-n-- ----------u-n-u-n-u-n--- u---------- u----------
Block 2	n----------n-n-----------n-n-n- ----------n-n-n-n-n-n-n- n---------- ----------
Block 3	----------n---n-n-n-n-n-n-n-n-- ----------n---n-n-n-n- ----------n----- ----------
Block 4	----------n-----n-n-n-n-n-n- ----------u----n-n-n-n- ----------n----- ----------
Block 5	----------n-n-n-n-n-n-n-n-n- ----------n-n-n-n-n--- ----------n----- ----------
Block 6	----------n-n-n-n-n-n-n-n-n- ----------n-n-n-n-n--- ----------n----- ----------
Block 7	----------n-n-n-n-n-n-n-n-n- ----------n-n-n-n-n--- ----------n----- ----------
Block 8	----------n-n-n-n-n-n-n-n-n- ----------n-n-n-n-n--- ----------n----- ----------
Block 9	----------n-n-n-n-n-n-n-n-n- ----------n-n-n-n-n--- ----------n----- ----------

Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH

Karthikeyan Bhargavan
INRIA
karthikeyan.bhargavan@inria.fr

Gaëtan Leurent
INRIA
gaetan.leurent@normalesup.org

***Abstract*—In response to high-profile attacks that exploit hash function collisions, software vendors have started to phase out the use of MD5 and SHA-1 in third-party digital signature applications such as X.509 certificates. However, weak hash constructions continue to be used in various cryptographic constructions within mainstream protocols such as TLS, IKE, and SSH, because practitioners argue that their use in these protocols relies only on second preimage resistance, and hence is unaffected by collisions. This paper systematically investigates and debunks this argument.**

We identify a new class of *transcript collision* attacks on key exchange protocols that rely on efficient collision-finding algorithms on the underlying hash constructions. We implement and demonstrate concrete credential-forwarding attacks on TLS 1.2 client authentication, TLS 1.3 server authentication, and TLS channel bindings. We describe almost-practical impersonation and downgrade attacks in TLS 1.1, IKEv2 and SSH-2. As far as we know, these are the first collision-based attacks on the cryptographic constructions used in these popular protocols.

Our practical attacks on TLS were responsibly disclosed (under the name SLOTH) and have resulted in security updates to several TLS libraries. Our analysis demonstrates the urgent need for disabling all uses of weak hash functions in mainstream protocols, and our recommendations have been incorporated in the upcoming Token Binding and TLS 1.3 protocols.

I. Introduction

Hash functions, such as MD5 and SHA-1, are widely used to build authentication and integrity mechanisms in cryptographic protocols. They are used within public-key certificates, digital signatures, message authentication codes (MAC), and key derivation functions (KDF). However, recent practical attacks on MD5 and almost-practical attacks on SHA-1 have led researchers and practitioners to question whether these uses of hash functions in popular protocols are still secure.

The first collision on MD5 was demonstrated in 2005 [38], and since then, collision-finding algorithms have gotten much better. Simple MD5 collisions can now be found in seconds on a standard desktop. In response, protocol experts reviewed the use of MD5 in Internet protocols such as Transport Layer Security (TLS) and IPsec [16], [15], [3]. Despite some disagreement on the long-term impact of collisions, they concluded that most uses of hash functions in these protocols were not affected by collisions. Consequently, MD5 continues to be supported (alongside newer, stronger hash algorithms) in protocols like TLS and IPsec.

In 2009, an MD5 collision was used to create a rogue CA certificate [36], hence breaking the security of certificate-based authentication in many protocols. A variant of this attack was used by the Flame malware to disguise itself as a valid Windows Update security patch [34]. Due to these high-profile attacks, there is now consensus among certification authorities and software vendors to stop issuing and accepting new MD5 certificates. Learning from the MD5 experience, software vendors are also pro-actively phasing out SHA-1 certificates, since collisions on SHA-1 are believed to be almost practical [35].

This leaves open the question of what to do about other uses of MD5 and SHA-1 in popular cryptographic protocols. Practitioners commonly believe that collisions only affect non-repudiable signatures (like certificates), but that signatures and MACs used within protocols are safe as long as they include unpredictable contents, such as nonces [16], [15].In these cases, protocol folklore says that a second preimage attack would be required to break these protocols, and such attacks are still considered hard, even for MD5.

Conversely, theoretical cryptographers routinely assume collision-resistance in proofs of security for these protocols. For example, various recent proofs of

NDSS '16, 21-24 February 2016, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2016.23418

TLS [17], [22], [11] assume collision-resistance even though the most popular hash functions used in TLS are MD5 and SHA-1. Whom shall we believe? Either it is the case that cryptographic proofs of these protocols are based on too-strong (i.e. false) assumptions that should be weakened, or that practitioners are wrong and collision resistance is required for protocol security.

This paper seeks to clarify this situation by systematically investigating the use of hash functions in the key exchanges underlying various versions of TLS, IPsec, and SSH. We demonstrate that, contrary to common belief, collisions *can* be used to break fundamental security guarantees of these protocols. We describe a generic class of attacks called *transcript collision* attacks, and detail concrete instances of these attacks against real-world applications. In particular, we demonstrate how a man-in-the-middle attacker can impersonate TLS 1.2 clients, TLS 1.3 servers, and IKEv2 initiators. We also show how a network attacker can downgrade TLS 1.1 and SSH-2 [39] connections to use weak ciphers. We implement proofs-of-concept exploit demos for three of these attacks to demonstrate their practicality, and provide attack complexities for the others. We believe that ours are the first hash collision-based attacks on the cryptographic constructions within these protocols.

We do not claim to have found all *transcript collision* attacks in these protocols; nor do we think that our attack implementations are the most efficient. Still, our results already provide enough evidence for us to strongly recommend that weak hash functions like MD5 and SHA-1 should be immediately disabled from Internet protocols. Partly due to recommendations by us and other researchers, these hash functions and other weak constructions based on them have been removed from the draft version of the TLS 1.3 protocol.

Outline Section II introduces transcript collision attacks on authenticated key exchange protocols. Section III outlines the state-of-the-art in collision-finding algorithms for MD5, SHA-1, and their concatenation. Section IV describe the TLS protocol, and Section V describes concrete attacks on various versions of TLS and three proof-of-concept demos. Section VI describes concrete attacks on IKE and SSH. Section VII summarizes the impact of our attacks and disclosure status. Section VIII concludes.

II. Transcript Collision Attacks on Authenticated Key Exchange

Authenticated Key Exchange (AKE) protocols are executed between two parties, usually called client and server or initiator and responder, in order to establish a shared session key that can be used to encrypt subsequent messages. A typical example is the SIGMA' protocol depicted in Figure 1. This protocol is a variant

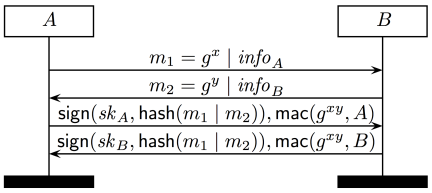


Fig. 1. SIGMA': A mutually-authenticated key exchange protocol

of the basic SIGMA (sign-and-mac) protocol from [21] which served as the inspiration for the key exchanges used in many protocols including IKE, OTR, and JFK.

In SIGMA', the initiator A first sends a message m_1 to B, consisting of Diffie-Hellman public value g^x, along with some protocol-specific parameters $info_A$ that may include, for example, a nonce, a protocol version, a proposed ciphersuite, etc. B responds with a message m_2 containing its own Diffie-Hellman public value g^y and some parameters $info_B$. A and B have now completed an anonymous Diffie-Hellman exchange and can compute the shared secret g^{xy} and use it to derive the session key. However, before using the session key, they authenticate each other by exchanging digital signatures over the protocol transcript $\mathsf{hash}(m_1|m_2)$ using their long-term signing keys (sk_A, sk_B). (Digital signature algorithms typically hash their arguments before signing them, and we have chosen to make this hashing explict in our presentation of SIGMA'.) By signing the transcript, A and B verify that they agree upon all the elements of the key exchange, and in particular, that a network attacker has not tampered with the messages. Finally, A and B also prove to each other that they know the session key g^{xy} by exchanging MACs computed with this key over their own identities.

Like other AKE protocols, SIGMA' aims to prevent message tampering, peer impersonation, and session key leakage, even if the network and other clients and servers are under the control of the adversary. Formally, authenticating the transcript guarantees *matching conversations*, that is, that the two parties agree on each others identity and other important protocol parameters.

Transcript Collision Attacks The alert reader will notice that SIGMA' *does not* in fact guarantee that A and B agree on the message sequence $m_1|m_2$; it only guarantees that they agree on the *hash* of this sequence. What if a network attacker were to tamper with the messages, so that A and B see different message sequences but the hashes of the two sequences is the same? In that case, the protocol will proceed to completion but the integrity and authentication guarantees no longer hold.

Figure 2 illustrates such an attack. The man-in-the-

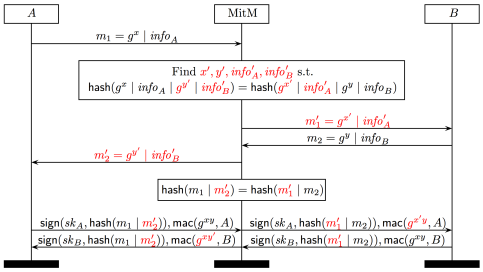


Fig. 2. Man-in-the-middle credential forwarding attack on SIGMA'. The attacker creates a *transcript collision* by tampering with the messages shown in red. At the end of the protocol, the client and server have seemingly authenticated each other, but the attacker knows both connection keys, and hence can read or write any data.

middle (MitM) intercepts messages sent between A and B. It sends its own message $m'_1 = g^{x'} | info'_A$ to B and it sends it own response $m'_2 = g^{y'} | info'_B$ to A. Suppose it can choose these messages such that the authenticated transcripts match:

$$\mathsf{hash}(m_1|m'_2) = \mathsf{hash}(m'_1|m_2)$$

We call this a *transcript collision*. Now, MitM can simply forward A's signature over this transcript to B and vice versa. A and B will accept the signatures since the hashed transcripts match and the signing keys are correct. However, the MitM knows the session keys $(g^{x'y}, g^{xy'})$ on both connections (since it knows x', y'). Hence, the MitM has fully hijacked both connections and can now send messages to B pretending to be A and to A pretending to be B. This is an *impersonation* attack that breaks peer authentication.

If the boundaries between the messages m_1 and m_2 are not clearly demarcated, there are a number of trivial attacks that can ensure that $m_1|m'_2 = m'_1|m_2$ with no need for hash collisions. In the examples of this paper, we will assume that each message (and each message field) is prefixed with its length, so that we can focus on attacks that rely on weaknesses in the hash function.

A Generic Transcript Collision The main challenge in implementing the attack in Figure 2 is that the MitM has to compute the messages m'_1 and m'_2 *after* receiving m_1 but *before* the responder has sent its response m_2. The feasibility of the attack depends on the contents and formats of these messages.

Suppose the responder B always sends the same message m_2 for every request; that is, it uses the same (static) Diffie-Hellman value g^y and same parameters $info_B$. (This situation occurs, for example in protocols like QUIC, where the server uses a static configuration.) In that case, after receiving m_1, the MitM can compute a transcript collision by finding $x', y', info'_A, info'_B$ such that $\mathsf{hash}(m_1|m'_2) = \mathsf{hash}(m'_1|m_2)$. The amount of work required to find such a collision depends on the hash function. As we will see in the next section, such collisions require $2^{N/2}$ work for hash functions that produce N bits. Hence, for MD5, such a collision would require the MitM to compute 2^{64} MD5 hashes, which may well be achievable by powerful adversaries.

A Chosen-Prefix Transcript Collision We now consider a more efficient attack that works even when B sends an unpredictable m_2 containing a fresh (ephemeral) Diffie-Hellman value g^y and a previously unknown $info_B$. However, we assume that the length of m_2 (M) is fixed and known to MitM. Moreover, suppose that in the second message of SIGMA', $info_B$ is allowed to have arbitrary length and arbitrary contents. That is, even if $info_B$ has junk data at the end, A will accept the message. Specifically, suppose that $info_B = len_B|data_B$ where $data_B$ is opaque data that will be ignored by A. (We will see several examples of such "collision-friendly" messages in TLS, IKE, and SSH.) Finally, we assume that the hash function uses the Merkle-Damgård construction [29], [7], so that it obeys the length extension property: if $\mathsf{hash}(x) = \mathsf{hash}(y)$ then $\mathsf{hash}(x|z) = \mathsf{hash}(y|z)$. (Strictly speaking, this property only holds when the lengths of x, y are equal and a multiple of the hash function block size.)

Under all these conditions, MitM can compute a transcript collision by finding two collision bitstrings C_1, C_2 of L_1 and L_2 bytes respectively, such that:

$$\mathsf{hash}(m_1|\underbrace{[g^{y'}|\overbrace{len'_B|C_1|-}^{info'_B}]}_{m'_2}) = \mathsf{hash}(\underbrace{[g^{x'}|\overbrace{C_2}^{info'_A}]}_{m'_1})$$

where $len'_B = L_1 + M$. Note that we have left empty space (written $-$) of size M bytes that still needs to be filled after C_1 in $info'_B$. As we will see in the next section, this kind of collision is called a *chosen-prefix* collision and is typically achievable with far less work than a generic collision attack. For example, a chosen-prefix collision in MD5 requires the MitM to compute about 2^{39} MD5 hashes, which takes only a few CPU hours.

After receiving m_1 from A and computing C_1, C_2, MitM now sends m'_1 to B. When B responds with m_2 (of size M bytes), MitM now stuffs m_2 at the end of $info'_B$ (in place of $-$) and sends m'_2 to A. Due to the length extension property, we have:

$$\mathsf{hash}(m_1|\underbrace{[g^{y'}|\overbrace{len'_B|C_1|m_2}^{info'_B}]}_{m'_2}) = \mathsf{hash}(\underbrace{[g^{x'}|\overbrace{C_2}^{info'_A}]}_{m'_1}|m_2)$$

That is, the MitM has obtained a transcript collision and the impersonation attack succeeds.

The attack here exploits hash collisions in combination with flexible protocol-specific message formats, and as we will see, this is one of the main novel *tricks* that we use to mount various attacks in this paper.

Other Transcript Collisions The transcript collisions described above are not the only attacks possible on such protocols. In some cases, MitM may not be able to use its own Diffie-Hellman values $g^{x'}, g^{y'}$ but it may still be able to tamper with the protocol parameters (e.g. ciphersuites) in $info'_A, info'_B$. In such cases, the MitM does not have full control over either connection (i.e. it cannot impersonate A or B) because it does know the session keys, but it may still be able to *downgrade* the protocol parameters to use weak, breakable ciphers.

In other cases, the message format may lend itself to simpler *common-prefix* collisions that require even less work than chosen-prefix collisions. Such collisions on MD5 can be found in seconds even on standard desktops. In the next section, we will discuss these different types of collisions in more detail (some technical details of previous results are given in the Appendix), and in the remainder of the paper, we will exploit them to mount transcript collision attacks on real-world protocols.

III. Hash Function Cryptanalysis

A hash function $H : \{0,1\}^* \to \{0,1\}^N$ maps arbitrary length binary strings to strings of N bits. Broadly speaking, a cryptographic hash function is expected to behave like a randomly selected function from the set of all functions from $\{0,1\}^*$ to $\{0,1\}^N$; building input/output values with specific properties should be as hard for H as for a random function. More concretely, a cryptographic hash functions should meet four goals:

1) **Preimage resistance:** Given a target value $\overline{H}$, it should be hard to find x such that $H(x) = \overline{H}$
2) **Second-preimage resistance:** Given an input x, it should be hard to find a second input $x' \neq x$ such that $H(x') = H(x)$
3) **Chosen-prefix collision resistance:** Given prefixes P and P', it should be hard to find a pair of values x, x' such that $H(P'|x') = H(P|x)$.
4) **Collision resistance:** For a hash function H, it should be hard to find a pair of inputs $x \neq x'$ such that $H(x') = H(x)$.

The expected security of a hash function is defined as the complexity of the best generic attack, *i.e.* the best attack that works on any hash function, without using any specific property of the design. For preimages or second-preimages, the best attack is a brute-force search: an adversary has to try about 2^N random inputs in order to find a preimage. However, for collisions, there is a generic attack with complexity $2^{N/2}$ because of the birthday paradox. If an adversary computes the images of a set of $2^{N/2}$ inputs, this defines about 2^N pairs of inputs, and there is a high probability that one of these pairs is a collision.

Generic collision attacks While a naive collision attack requires to store $2^{N/2}$ images of the hash function, it is possible to mount a parallel and memory-less attack with a very small overhead [37]. This generic collision attack is very powerful: it can use meaningful messages, and can easily be used for chosen-prefix collisions (see details in Appendix).

Concatenation To strengthen protocols against collisions in any one hash function, it may be tempting to use a combination of two independent hash functions. For example, TLS versions up to 1.1 use a concatenation of MD5 and SHA-1. While the output length of this construction is 288 bits, it does not offer the security of a 288-bit hash function. In particular, Joux described a multi-collision attack that breaks the concatenation of two hash functions with roughly the same effort as breaking the strongest one of the two [18].

Shortcut collision attacks In the last decade, hash function cryptanalysis has been a very active research area, and more efficient attacks have been discovered on widely used hash functions. The (estimated) complexity of the best attacks currently known against MD5 and SHA-1 are the following:

MD5	Common-prefix collision: 2^{16} [36]
	Chosen-prefix collision: 2^{39} [36]
SHA-1	Common-prefix collision: 2^{61} [35]
	Chosen-prefix collision: 2^{77} [35]
MD5 | SHA-1	Common-prefix collision: 2^{67} [18]
	Chosen-prefix collision: 2^{77} [18]

Shortcut collision attack usually return messages with random-looking blocks that are not controlled by the adversary. This makes it harder to use these messages in a real attack, but we will see that in many cases we can still have meaningful messages by stuffing the random blocks in non-significant sections.

Implementation of attacks Since generic collision attacks can be easily parallelized and require little memory, they can efficiently be implemented in GPUs. In particular, an attack against MD5 require 2^{64} computations. This is well within reach for a motivated adversary: it would cost around $\$165\,000$ on Amazon EC2 (using a spot price of 8 ¢/h for a g2.2xlarge instance doing 2.5 GH/s). Dedicated hardware would be significantly more efficient, but require a large investment. As

a point of comparison, the current Bitcoin network is able to compute up to 2^{59} SHA-256 hashes per second.

We have implemented this attack against the 96-bit MAC used for the Finished message of TLS 1.1. Our demo took 20 days using four Tesla K20Xm GPUs, which is comparable to the expected time we can derive from hash function benchmarks.

For a chosen-prefix collision, an important part of the computation is spend constructed differential paths, and this is much harder to parallelize on GPU. We used the HashClash software [33] by Marc Stevens to perform this computation. Stevens *et al.*'s estimate that the chosen-collision attack should require 2^{39} hash computations, or 35 core-hours [36]. In order to build the collision as fast as possible, we modified the software to take better advantage of parallelism. The hashclash software spends most of its time building differential paths, with a forward step, a backward step, and a connection step. We realized that the backward step uses a limited number of potential starting points, and we precomputed the results for all possible starting points. In addition, we merged the forward and connection steps, in order to avoid the serialization and deserialization of the result. With these optimisations, we can build a chosen-prefix collision in one hour with a 48 cores machine, using a few gigabytes of RAM (the original code required at least 3 hours). We believe the time can be further reduced, but this will require a significant rewrite of the hashclash software to allow parallelism across several machines, or to rewrite it for GPUs.

IV. The TLS Handshake Protocol

The Transport Layer Security protocol (TLS) [8] is perhaps the most widely used secure channel protocol. Many versions of TLS are used on the Internet; the latest released version is TLS 1.2 [8], while TLS 1.3 [9] is currently undergoing standardization at the IETF.

Figure 3 depicts a typical handshake in TLS (in versions 1.0 to 1.2). The client first sends a hello message `CH` that contains a fresh random client nonce n_c and various protocol parameters ex_c, including the protocol version, supported list of ciphersuites, and various protocol extensions. Each extension is prefixed by its length and can contain a payload of up to 2^{16} bytes. Notably, the client hello may include extensions that the server does not understand or support, and the server will ignore them.

The server responds to the client hello with a series of messages (from `SH` to `SHD`). The server hello `SH` contains a fresh server nonce n_s and parameters ex_s, including the server's chosen version, ciphersuite, and protocol extensions. In most ciphersuites, the server then sends its public-key certificate `SC`. In Ephemeral Diffie-Hellman (DHE) ciphersuites, `SC` is followed by a server

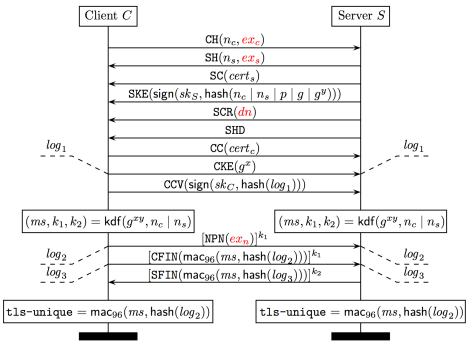


Fig. 3. TLS 1.2: A mutually-authenticated DHE handshake. Fields shown in red indicate parts of the handshake that can contain arbitrary-length opaque data (useful for stuffing collision blocks). Handshake transcripts (log_1, log_2, log_3) refer to the concatenation of all messages up to (and including) the current one. Messages `SCR`, `CC`, `CCV` are optional and only appear when client certificate authentication is used. `NPN` is optional and only appears when the client and server support the next-protocol-negotiation extension. The `tls-unique` channel binding is a connection identifier that may be used by applications to bind user authentication tokens, such as cookies and passwords, to the underlying TLS channel to prevent credential forwarding.

key exchange message `SKE` that contains an ephemeral public value g^y along with a description of the Diffie-Hellman group chosen by the server, including the prime p and generator g. The server signs these values to protect then from tampering and to prove that it knows the private key (sk_S) for the certificate:

$$\mathsf{sign}(sk_S, \mathsf{hash}(n_c|n_s|p|g|g^y))$$

The signature and hash algorithm used for this signature is chosen by the server based on its certificate as well as the supported algorithms indicated by the client within an optional `signature-algorithms` extension in the client hello. In TLS versions before 1.2, the hash algorithm was fixed to be MD5 | SHA-1 but TLS 1.2 allows clients and servers to choose any hash algorithm they both support (MD5, SHA-1, SHA-256, etc.) Hence in TLS 1.2, each digital signature is prefixed with identifiers for the chosen signature and hash algorithm.

If the server wants the client to authenticate itself with a public-key certificate, it then sends a certificate request message `SCR` indicating the certificate types and signature algorithms it supports, as well as an optional list of distinguished names dn for the client certification authorities that it trusts. As with hello extensions, each distinguished name can be 2^{16} bytes long and can contain arbitrary data that the client will ignore if it does not recognize the name. The server's message flight then ends with the server hello done message `SHD`.

The client then sends its own certificate `CC` if the

server asked for it, and its own Diffie-Hellman key share g^x in a client key exchange message CKE. If the client sent a certificate, it must prove that it knows the private key sk_C by sending a client certificate verify CCV message with a signature over the full message log up to this point in the protocol:

$$\mathsf{sign}(sk_C, \mathsf{hash}(\underbrace{\texttt{CH}|\texttt{SH}|\texttt{SC}|\texttt{SKE}|\texttt{SCR}|\texttt{SHD}|\texttt{CC}|\texttt{CKE}}_{log_1}))$$

At this point, the client and server both derive a session master secret ms and authenticated encryption keys for both directions (k_1, k_2). The client sends a change cipher spec message to indicate that the subsequent messages it sends will be encrypted (with k_1.) This message is not technically part of the handshake protocol and does not appear in the authenticated transcript, and so it is not shown in Figure 3.

If the client and server both indicate support for the next-protocol-negotiation extension [24] in their hello messages, the client then sends an encrypted extensions message NPN containing a selected application layer protocol (e.g. `http/1.1` or `spdy/3`). The protocol name is ASCII-encoded and then padded to the nearest multiple of 32 bytes (to avoid leaking information via the encrypted message length.)

The client then sends an encrypted finished message CFIN containing a MAC of the full handshake log log_2 using the master secret ms. In TLS 1.0 and 1.1, this MAC is computed using a combination of HMAC-MD5 and HMAC-SHA-1, whereas in TLS 1.2, it uses HMAC-SHA-256. In all these versions, the result of the MAC is then truncated to 12 bytes (96 bits):

$$\mathsf{mac}_{96}(ms, \mathsf{hash}(\underbrace{\texttt{CH}|\texttt{SH}|\texttt{SC}|\texttt{SKE}|\texttt{SCR}|\texttt{SHD}|\texttt{CC}|\texttt{CKE}|\texttt{CCV}|\texttt{NPN}}_{log_2}))$$

When a server receives CFIN, it verifies that the client agrees with it on the full message log and on the master secret. It responds by sending its own change cipher spec message to turn on encryption and a server finished message SFIN that contains a 96-bit MAC over the full handshake log log_3 using the master secret ms.

At the end of the handshake, both client and server have authenticated each other, proved knowledge of the master secret, and agreed upon the message log. They can now start encrypting application data to each other using the connection keys (k_1, k_2).

In most common TLS usage scenarios, clients are not authenticated using certificates. The handshake authenticates only the server and the client-side user is authenticated within the application using a challenge-response protocol based on a password or some other bearer token (e.g. HTTP cookie). Such application-level authentication protocols are known to be vulnerable to a general class of *credential forwarding* attacks unless the application-level credential is channel bound to the TLS connection (e.g. see [5]. In such attacks, a client C connects to a malicious server M and authenticates with some credential over TLS, but M forwards the authentication message over another TLS channel to S, thereby logging in as C at S. The attack is prevented if the authentication protocol embeds a unique identifier for the underlying TLS channel, so that a message sent over one channel cannot be forwarded over another. One such identifier, called `tls-unique`, defined in [2], uses the contents of the CFIN message as a unique identifier for the TLS connection. This `tls-unique` channel binding is used by a number of emerging application-level authentication protocols, such as SCRAM [28], FIDO [14], and Token Binding [32], specifically to avoid credential-forwarding attacks.

V. Transcript Collision Attacks on TLS

As we saw in the previous section, TLS uses a variety of hash constructions to implement key security mechanisms like client and server authentication, handshake integrity, and channel binding. We now demonstrate weaknesses in these constructions and show how they can be exploited to mount practical transcript collision attacks on real-world clients and servers.

A. Breaking TLS 1.2 Client Authentication using a Chosen-Prefix Transcript Collision

Suppose a client C uses the same certificate to connect to two different servers A and S. We show that if A is malicious, it can force C to create a signature (in CCV) that A can use to impersonate C at S, as depicted in Figure 4. Here, A acts as a man-in-the-middle between C and S. Note, however, that A uses its own certificate $cert_a$ and does not rely on knowing any long-term secrets belonging to C or S.

Recall that the client signs the transcript $\mathsf{hash}(log_1^c)$; so the key idea of the attack is to compute a collision between this client-side transcript and the server-side transcript $\mathsf{hash}(log_1^s)$, even though the two connections see different message sequences. When the MitM A receives a client hello from C, it responds with its own hello SH′, certificate SC′, key exchange SKE′. It then initiates a connection with the server S by sending a carefully crafted client hello CH′. A now runs both connections in parallel. It will receive a hello SH, certificate SC, key exchange SKE, and certificate request SCR from S. We assume that the length of these messages SH|SC|SKE|SCR is fixed ($= M$) and is known in advance.

Note that A needs to choose CH′ before it receives any messages from S. A can compute CH′ and SCR′ as follows. A uses a chosen-prefix collision to find two bit-strings (C_1, C_2) of length L_1 and L_2 bytes respectively

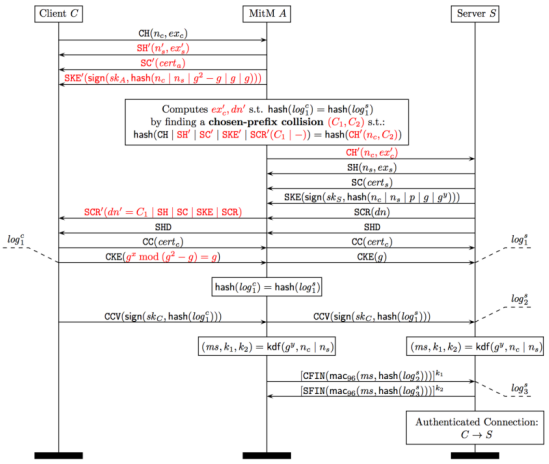


Fig. 4. Man-in-the-middle client signature forwarding attack on TLS 1.2. The client C connects to a malicious server A and offers to authenticate with its certificate $cert_C$. The attacker A computes a chosen-prefix collision on the client signature transcript $\mathsf{hash}(log_1)$, and uses it to impersonate the client at a different server S. Messages that the attacker controls are labeled in red. A sends a bogus Diffie-Hellman group $(k^2 - k, k)$ to C; we use $k = g$ here for simplicity.

such that C_1 appears within the last distinguished name dn' in $\mathtt{SCR}'$ and C_2 appears within the last extension ex'_c in $\mathtt{CH}'$:

$$\begin{aligned} & \mathsf{hash}(\mathtt{CH}|\mathtt{SH}'|\mathtt{SC}'|\mathtt{SKE}'|\mathtt{SCR}'(\underbrace{C_1}_{dn'}|-)) \\ = \; & \mathsf{hash}(\mathtt{CH}'(n_c, \underbrace{C_2}_{ex'_c})) \end{aligned}$$

Furthermore, we set the length of dn' in $\mathtt{SCR}'$ to be $L_1 + M$, so that it still has M bytes (denoted by $-$) that need to be filled in after C_1.

Now, A sends $\mathtt{CH}'$ to S, receives $\mathtt{SH}|\mathtt{SC}|\mathtt{SKE}|\mathtt{SCR}$ in response, and stuffs these messages into the remaining M bytes in $\mathtt{SCR}'$ and sends it to C. At this point the hash of the message transcripts in the two connections coincide, assuming that the hash function satisfies the length extension property:

$$\begin{aligned} & \mathsf{hash}(\mathtt{CH}|\mathtt{SH}|\mathtt{SC}'|\mathtt{SKE}'|\mathtt{SCR}'(\underbrace{C_1|\mathtt{SH}|\mathtt{SC}|\mathtt{SKE}|\mathtt{SCR}}_{dn'})) \\ = \; & \mathsf{hash}(\mathtt{CH}'(n_c, \underbrace{C_2}_{ex'_c})|\mathtt{SH}|\mathtt{SC}|\mathtt{SKE}|\mathtt{SCR}) \end{aligned}$$

From this message onwards, the hash of the handshake log in both connections will remain the same. A then forwards the sever hello done $\mathtt{SHD}$ to C. In response, C sends a certificate $\mathtt{CC}$, a key exchange $\mathtt{CKE}$, and a certificate verify $\mathtt{CCV}$ that contains a signature over the transcript $\mathsf{hash}(log_1^c)$ which is now the same as $\mathsf{hash}(log_1^s)$. A simply forwards these messages to S, pretending to be C, and S accepts these messages.

Controlling the master secret Even though S has accept C's certificate on its connection with A, A cannot complete the connection unless it knows the master secret on its connection with S. The master secret is computed from g^{xy} so A needs to know the x corresponding the g^x that C sent in its key exchange message $\mathtt{CKE}$. In order to accomplish this task, we rely on a key forcing attack in the DHE handshake.

When A sends $\mathtt{SKE}'$ to C, it does not send a valid Diffie-Hellman group (p, g). Instead, it chooses an arbitrary public value $k = g^{x'}$ and sets $p = k^2 - k$ and $g = k$. This p value is clearly not a prime, and it has the property that no matter what private value x is generated by C, we will have $g^x \bmod p = k$. Hence, by choosing such a bogus Diffie-Hellman group, A can force C to send a $\mathtt{CKE}$ with a public value that it controls.

To complete the attack, we assume that S always uses the same Diffie-Hellman group (p, g). A chooses some x' and sets $k = g^{x'} \bmod p$. It then sends $\mathtt{SKE}'$ to C with the bogus group $(k^2 - k, k)$ and the public value k. Now, the $\mathtt{CKE}$ sent by C will contain k, and A will forward it to S. A will then forward C's signature $\mathtt{CCV}$ as usual. The master secret between A and S will be derived from $g^{x'y} \bmod p$, but A knows x' and hence can compute this value. Consequently, A can complete the handshake and impersonate C at S.

We observe that the attack here relies on the client not validating the Diffie-Hellman groups it receives from the server. From our experiments, we find that most TLS libraries do not validate the groups they receive in the server key exchange, probably because checking for primality is expensive. In some libraries, the value $k^2 - k$ is rejected because it is an even number. In those cases, we find that we can use $p = k^2 - 1$ and with 50% probability, the client will compute $g^x = k$, allowing the attack to succeed. This weakness in TLS-DHE has been noted before [6] and a new protocol extension aims to fix it by allowing only well-known Diffie-Hellman groups [12]. However, an optional extension cannot prevent our attack scenario, since A could always pretend to not support the extension and mount the attack anyway.

Note that the attack only relies on DHE between C and A; the connection between A and S can use ECDHE or RSA and the attack would still work. In other words, such transcript collisions can also be used to mount cross-protocol attacks in the sense of [26].

Attack Complexity The transcript collision attack requires A to compute a chosen-prefix collision for the hash function used in the client signature. In TLS

versions before 1.2, the default hash function is a concatenation of MD5 and SHA-1 and hence requires computing 2^{77} MD5 and SHA-1 hashes. In TLS 1.2, if the signature uses SHA-1, the cost is 2^{77} hashes. Remarkably, TLS 1.2 also allows RSA-MD5 signatures, and for such signatures, the cost of the collision is only 2^{39} MD5 hashes. Below, we describe our proof-of-concept implementation that relies on RSA-MD5.

Note that these cost estimates are per-connection because the collision needs to be computed once for each client nonce n_c. Usually, these nonces are generated with a strong random number generator. However, in some cases the client random can become predictable due to implementation bugs (e.g. see CVE-2015-0285 in OpenSSL). We also observe that it is commonly believed that these nonces only need to be unique, not unpredictable. For example, the OpenSSL library uses `RAND_pseudo_bytes` to generate the client and server random, whereas it uses `RAND_bytes` to generate other key material; the former succeeds even when there is not enough entropy in the system. If the client nonce were predictable, or if it were to be repeated with high frequency, the collision can be computed offline at leisure, making SHA-1 collisions almost feasible. Even though our attack below does not rely on predictable nonces, it offers yet another justification for the need for strongly random nonces in TLS.

Implementing a Proof-Of-Concept To implement the attack, we need a client that is willing to sign with RSA-MD5 and a server that is willing to accept such signatures. We found a number of TLS libraries that support RSA-MD5 client signatures, including certain versions of OpenSSL, GnuTLS, Oracle and IBM Java, and BouncyCastle. (See Section VII for more details.) In particular, all major Java web application servers and the default TLS servers on Red Hat Enterprise Linux (6 and 7) accept RSA-MD5 signatures.

For our demo, we set up a man-in-the-middle attack between a standard Java HTTPS client and a Java HTTPS server (with default configurations.) The MitM implements Figure 4. In order to setup the collision while preserving the TLS message formats, the attacker needs to carefully set the length fields in various places in $\texttt{CH}'$ and $\texttt{SCR}'$. For example, in $\texttt{CH}'$ it needs to set consistent lengths for the full hello message, for the extensions field, and for the last extension. Furthermore, the MitM needs to make sure that the two prefixes have a length that is a multiple of the MD5 block size (512 bits). To achieve this, we fill up the last extension in $\texttt{CH}'$ and the last distinguished name in $\texttt{SCR}'$ with enough zero bytes until the prefixes are block-aligned.

As explained in Section III, the chosen-prefix collision can be be computed in one hour on a 48 core workstation using a modified version of the hashclash software [33]. In our demo, A accepts the client hello and then keeps the client-side TLS connection alive until a collision has been found. Most TLS connections can be kept alive by sending regular warning alerts; Java clients are willing to keep the connection open indefinitely. Keeping the client waiting for an hour is not always practical, but we note that some unsupervised TLS clients (such as git) are used to perform long-running connections to web APIs, and long connection times may not be noticed. In any case, the collision search scales well with computational power and can be significantly sped up by a powerful adversary.

Once the collision has been found, A connects to S to completes the attack and is able to impersonate C at S and read and write data that only C should have access to. Hence, the demo shows that A is able to break TLS 1.2 client authentication between mainstream TLS clients and servers. The precise handshake traces exhibiting the collision are available from our website.

B. Breaking TLS 1.2 Server Authentication using a Generic Transcript Collision

The key to our attack above on TLS 1.2 client authentication is that the client is willing to sign the hash of the full message log, and the format of various TLS messages is flexible enough to allow the attacker to stuff meaningless collision blocks and server-side messages into them. A similar chosen-prefix transcript collision attack would not work on TLS 1.2 server authentication because the server signature transcript does not contain flexible-size elements.

In DHE handshakes, the signature covers only the client and server nonces and the server's Diffie-Hellman key share: $\mathsf{sign}(sk_S, \mathsf{hash}(n_c|n_s|p|g|g^y))$. So, the only part of the signed value that the attacker may control is the client nonce n_c which is fixed-length (32 bytes), half the size of one MD5 block.

This prevents the use of shortcut collision attacks against MD5, but generic attacks based on the small 128-bit MD5 hash length are still possible, and not too far from being practical.

Collecting and storing signatures. To mount a transcript collision attack on TLS 1.2 server authentication, an attacker first has to collect a large number, say 2^x, of RSA-MD5 signatures signed by the server. The attacker may do this by passively observing RSA-MD5 connections to the server, but since such connections may be rare, it may have to actively connect to the server to obtain a sufficient number of signatures. Once these signatures (and the corresponding hashes) have been collected and stored, the attacker can impersonate the server to any client.

Upon receiving a client hello message including the client nonce n_c, the attacker chooses a DH secret y' and computes the MD5 hashes of the transcripts $n_c|n_s|p|g|g^{y'}$ for a series of random server nonces n_s, until the hash matches a value that was collected previously. Finding this collision requires the attacker to compute about 2^{128-x} MD5 hashes and then look them up in the stored signature database. When a match is found, the stored signature can be used by the attacker to forge the server's SKE message for the current connection, and hence impersonate the server.

The complexity of this attack on TLS 1.2 server authentication is therefore 2^{128-x} MD5 hashes per connection, in addition to 2^x connections performed before-hand, and 2^x storage. The attacker can trade-off between these costs—the more signatures he can collect, the less he has to compute per connection. For example, if it is feasible to collect, store, and search through 2^{64} signatures, then the per-connection cost is 2^{64} hashes. Although we have described the attack in terms of MD5, a similar but more expensive attack can be mounted on RSA-SHA1 server signatures, which would require 2^{160-x} computation per-connection.

Practical Impact of the Attack. Both the precomputation and per-connection cost of the attack is currently out of reach for academic researchers, but might be within the capabilities of well-resourced adversaries.

A prerequisite for the attack is to find servers that would be willing to sign their SKE messages with RSA-MD5. Internet-wide scans show that about 31% of the Alexa top 1 million websites support RSA-MD5 signatures.[1] This subset includes popular websites hosted by Akamai, such as microsoft.com.

A second question is whether TLS clients would accept RSA-MD5 signatures. Most popular web browsers and TLS libraries do not offer RSA-MD5 as one of the supported signature algorithm in the client hello. This might lead one to believe that they would not accept RSA-MD5 server signatures. However, we found and reported security bugs in NSS (the library used by Firefox and some versions of Chrome), GnuTLS (used in curl and git), and BouncyCastle; these libraries (and applications that rely on them) incorrectly accept RSA-MD5 signatures even if they have been explicitly disabled. For example, Firefox will accept an RSA-MD5 signature from a website, even though it is not supposed to. Furthermore, other TLS libraries such as versions of OpenSSL (up to version 1.0.1e), mbedTLS, and Java routinely offer and accept RSA-MD5 signatures.

Conseqently, if an attacker has the resources to achieve the server impersonation attack, a large number of TLS clients would be affected. To err on the safe side, we recommend that TLS libraries should immediately disable all MD5-based signatures.

[1] https://securitypitfalls.wordpress.com/2015/12/07/november-2015-scan-results/

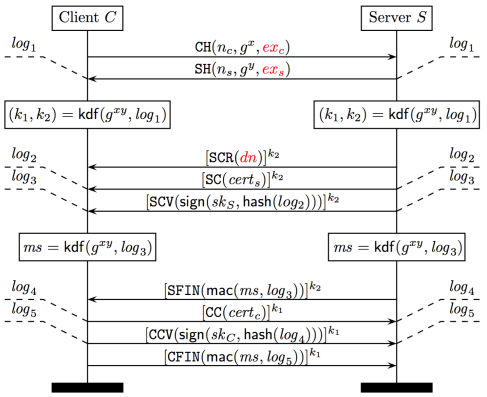


Fig. 5. TLS 1.3: A server-authenticated 1-RTT (EC)DHE handshake based on draft 10 of the specification. The client and server send their key shares within the hello messages and all subsequent handshake messages are encrypted. The server signs the current handshake transcript $\mathsf{hash}(log_2)$ in a new SCV message.

Exploiting predictable nonces and keys. We observe that the precomputation in the above attack can be avoided if the server uses a predictable nonce n_s and predictable DH parameters p, g, g^y. In this case, the attacker only has to perform 2^{64} computations online. How realistic is this assumption? Many TLS implementations allow DH keys to be reused; in OpenSSL, for example, keys are reused by default unless the application sets the SSL_OP_SINGLE_DH_USE flag. For such servers, the parameters are clearly predictable.

That leaves the server nonce, and as we noted for clients, a bug in the use of the random number generator could lead to predictable nonces. Alternatively, the server may support a recent TLS variant called Snap-Start [23] that allows the client (and hence the attacker) to choose the server nonce. That said, we do not know of any deployed TLS 1.2 implementation that uses predictable nonces, but this section serves as a warning to implementors that strongly random nonces are needed in TLS, and not just for preventing replays.

C. Breaking TLS 1.3 Client and Server Authentication using a Chosen-Prefix Transcript Collision

From the viewpoint of transcript collisions, TLS 1.2 server signatures may seem stronger than client signatures, but not signing enough leads to other security problems. For example, the server becomes vulnerable

to cross-protocol attacks [26] and to downgrade attacks like Logjam [1]. In response to such attacks, the new design of TLS 1.3 requires the server to sign the full handshake log, including the negotiated parameters.

Figure 5 illustrates the standard one-round-trip (1-RTT) message flow in the current draft (version 10) of the TLS 1.3 specification. In comparison to TLS 1.2, this protocol flips the order in which the DH key shares are sent, so that the handshake can complete in one round trip. The key shares are sent within extensions in the hello messages CH and SH. The server no longer sends an SKE message. Instead, it sends a new server certificate verify message SCV just before the finished message. The SCV contains a signature over the hash of the full message log up to this point (log_2). Another departure from TLS 1.2 is that all handshake messages after SH are encrypted, in order to protect the privacy of the client and server certificates from passive attackers.

We demonstrate a chosen-prefix transcript collision on TLS 1.3 that breaks both client and server signatures, enabling a full man-in-the-middle attack on the protocol. The attack is similar in spirit to the one on TLS 1.2 client signatures; we use the flexible formats of the client and server hello messages to create a trascript collision immediately after the server hello SH.

The client C wants to connect to S, but its messages are intercepted by a network attacker A. After A receives the client's CH, it sends its own CH′ to the server, receives the servers SH, and sends its own SH′ to the client. A now knows the Diffie-Hellman shared secrets on both connections, and it has chosen CH′ and SH′ such that $\mathsf{hash}(\mathtt{CH}|\mathtt{SH}') = \mathsf{hash}(\mathtt{CH}'|\mathtt{SH})$. Consequently, A can now simply forward all hanshake messages between C and S, and both client and server authentication will succeed. A will need to decrypt and reencrypt these messages, but it can do so because it knows the encryption keys on both connections. More importantly, once the handshake is complete, A can read and tamper with application data in both directions.

To compute CH′ and SH′, A needs to find a chosen-prefix collision C_1, C_2 of length L_1 and L_2 bytes respectively such that C_1 appears within the last extension of of SH′ and C_2 appears as the last extension of CH′:

$$\mathsf{hash}(\mathtt{CH}|\mathtt{SH}'(n_s, g^{y'}, \underbrace{C_1|-}_{ex'_s})) = \mathsf{hash}(\mathtt{CH}'(n_c, g^{x'}, \underbrace{C_2}_{ex'_c}))$$

Suppose we know that the server S will respond to CH′ with a server hello message SH of known length M. Then in SH′, we set the length of ex'_s to $L_1 + M$ so that there is room for M more bytes after C_1. Once A receives SH from S, it stuffs this message within this extra space in SH′ and sends it to C. Hence, after the server hello, the handshake transcripts at the client and server have the same hash. Moreover, due to the length extension property of the hash function, all subsequent handshake hashes collide. So, A can forward S's signature in SCV to C and C's signature in CCV to S, and both will be accepted, even though the DH keys have been tampered with by a man-in-the-middle.

Implementing a Proof-Of-Concept Up to draft 7, the TLS 1.3 specification explicitly allowed RSA-MD5 signatures. We wrote a proof-of-concept attack demo based on our own simple prototype implementation of TLS 1.3 that signs with RSA-MD5. As with TLS 1.2 client authentication, we found the chosen-prefix collision in roughly one hour on a single workstation.

As we observed when discussing TLS 1.2, a large number of TLS servers and clients support RSA-MD5 signatures. Consequently, we believe that if TLS 1.3 draft 7 were to be implemented today, it is quite likely that many of its clients and servers would be vulnerable to our man-in-the-middle attack. However, this attack vector was removed from TLS 1.3, at least partly due to our findings, when draft 8 of the protocol explicitly deprecated MD5-based signatures.

D. Downgrading TLS 1.0-1.1 to Weak Ciphersuites using a Chosen-Prefix Transcript Collision

In TLS, the integrity of the handshake depends upon the MACs exchanged in the Finished messages. If these MACs were broken, the attacker would be free to modify the hello messages to downgrade the connection to an old protocol version or weak ciphersuite, or to delete important extensions such as the renegotiation indication countermeasure [11].

Recall that the Finished MACs are computed over the *hash* of the full handshake trascript ($\mathsf{hash}(log_2)$ and $\mathsf{hash}(log_3)$ in Figure 3). In TLS 1.0 and 1.1, this hash function is the concatenation of MD5 and SHA-1. As we saw in Section III, a chosen-prefix collision on this construction can be computed with 2^{77} work. We find a man-in-the-middle transcript collision attack on server-authenticated TLS 1.1 that is similar to the TLS 1.3 attack. A network attacker modifies the client and server hellos so that the handshake hashes collide immediately after these two messages; the rest of the handshake is left unchanged. The client authenticates the server and the handshake completes successfully, and although the attacker does not know the master secret, it can downgrade the connection to use any weak algorithm that both the client and server support, but prefer not to use, such as an EXPORT ciphersuite [1], or a weak encryption algorithm like RC4.

A similar transcript collision attack appears in DTLS 1.0, a UDP-based variant of TLS 1.1. In DTLS, the attack can be made even more efficient by exploiting its cookie mechanism. In response to a client hello CH,

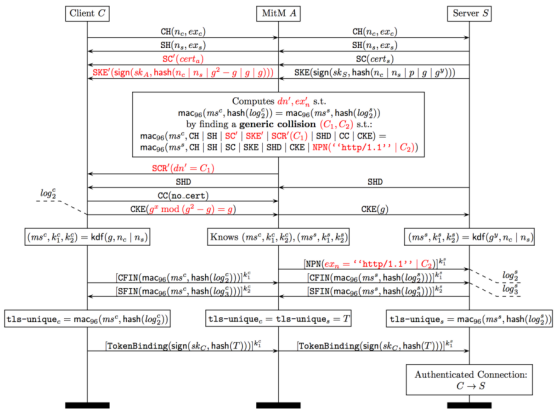


Fig. 6. Man-in-the-middle credential forwarding attack on `tls-unique` channel binding. The attack uses a transcript collision to impersonate the client to the server.

a DTLS server can send a hello verify request message `HVR` containing a cookie. The client is meant to restart the handshake by sending the exact same client hello message but with this cookie included in it. Since the `HVR` is not authenticated, the arbitrary-length cookie field allows any network attacker to inject data into the transcript, after a known prefix of a fixed length. This allows the chosen-prefix attack to be transformed to an *almost* common prefix attack, similar to the cookie-based attack on IKEv2 in Section VI-A.

E. Breaking the `tls-unique` Channel Binding using a Generic Transcript Collision

Suppose an application-level authentication protocol at C binds its login credential to the `tls-unique` channel binding [2], so that when the credential is sent from C to A, it cannot be used by A at S. We demonstrate how the attacker A could use a generic collision attack to break this protection.

Figure 6 depicts the attack. It follows the general pattern of the TLS 1.2 client authentication attack, except that it relies on a collision on the transcript MAC in the client finished message, rather than a collision in the hash function. The client C connects to the MitM A who then opens a new connection to S. The attacker sends a $\mathtt{SKE}'$ to C that contains a bogus group $(k^2 - 1, k)$, thereby forcing the client to send $k^x \bmod (k^2 - 1) = k$ in its client key exchange `CKE`. On the server side, the attacker can send its own $\mathtt{CKE}'$ containing any Diffie-Hellman value. Hence, the MitM knows the master secrets ms^c, ms^s and connection keys on both connections.

The goal of the attacker is to make sure that the contents of the client finished message (i.e. the `tls-unique`) coincide on both connections:

$$\mathsf{mac}_{96}(ms^c, log_2^c) = \mathsf{mac}_{96}(ms^s, log_2^s)$$

The attacker can use any controlled part of the transcript, but we will set things up carefully so that he can compute the collision as late as possible, in order to reduce the size of the messages to hash. More precisely, we use the certificate request $\mathtt{SCR}'$ on the client-side and the `NPN` message on the server side, which are sent when all other messages in the transcript are already fixed. The attacker uses C_1 as the last distinguished name in $\mathtt{SCR}'$ and C_2 as the padding in the `NPN` message (after the protocol name "`http/1.1`"), and computes (C_1, C_2) such that the MAC coincides. Once this collision is found, the MitM sends these two messages on the corresponding connections and completes the handshakes. A can then impersonate C at S by forwarding any application-level channel-bound credentials sent by C (for A) to S.

Implementing a Proof-Of-Concept We implemented a man-in-the-middle attacker to demonstrate the attack. We used an OpenSSL client as C and the main Google website as S, since this website supports the next-protocol-negotiation protocol extension. After receiving the client hello `CH` from the client and the server hello done `SHD` from the server-side, the MitM runs a generic collision search to compute $\mathtt{SCR}'$ and `NPN`.

For the collision search, we implemented the TLS PRF mac_{96} function using the CUDA framework for NVIDIA GPUs. In TLS versions up to 1.1, this construction is built using MD5 and SHA-1; in TLS 1.2 the construction uses SHA-256. However, the strength of the hash function is immaterial because what we are attacking is the truncated 96-bit MAC. The underlying hash function does not matter. Following the analysis explain in Section III, it should require about 2^{48} computations on average to get a collision.

Our implementation run at 160 MH/s for TLS 1.1 and 113 MH/s for TLS 1.2 on a Tesla K20Xm GPU. This is comparable to the expected speed we can derive from benchmarks of MD5, SHA-1 and SHA-256 on this GPU.It took 20 days to find a collision for TLS 1.1, using four Tesla K20Xm GPUs. Our demo evaluated the PRF about $2^{49.9}$ times, which is rather unlucky: it should take half that number on average. We note that the generic collision attack is completely parallelizable and hence the time for finding a collision can be brought down to an arbitrarily small number by throwing enough computational power at it. Using Amazon EC2, this should cost about $140 for TLS 1.1, and $200 for TLS 1.2. The transcripts are available on our website.

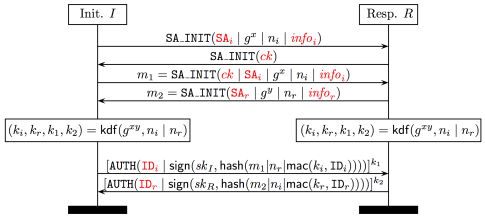


Fig. 7. IKEv2: A mutually-authenticated key exchange. Message parts colored in red can have arbitrary length.

Truncated HMAC is not collision-resistant A more general lesson to be taken from our attack on `tls-unique` is that there are many uses of HMAC in cryptographic protocols that are not protected from collisions in the underlying hash function. For example, although HMAC-MD5 may be a good MAC, it is not collision-resistant when the key is known to the attacker. Similarly, when HMAC-SHA256 is truncated to 96 bits, it may still be a good MAC, but it is certainly not a good hash function (since collisions can still be found in 2^{48} steps). Consequently, when inspecting the use of hash functions in Internet protocols, it would be a mistake to assume that all uses of HMAC are safe; it is important to look both at the mechanism and its intended security goal. In some cases, we may need HMAC to be both a MAC and a collision-resistant hash function.

VI. Transcript Collisions in IKE and SSH

Although the bulk of this paper has focused on collisions in TLS, similar attacks apply to other mainstream protocols like IKEv1, IKEv2, and SSH. We describe two exemplary attacks here.

A. Breaking IKEv2 Initiator Authentication using a Precomputed Common-Prefix Transcript Collision

Figure 7 depicts the IKEv2 authenticated key exchange protocol, which is similar to the SIGMA' protocol discussed in Section II. The initiator first sends an `SA_INIT` request containing its Diffie-Hellman value g^x, nonce n_i, and proposed cryptographic parameters $\mathtt{SA}_i, info_i$. The responder replies with its own public value g^y, nonce n_r and parameters $\mathtt{SA}_r, info_r$. Alternatively, the responder may send a cookie ck, thereby asking the initiator to restart the protocol by sending the same `SA_INIT` request but with ck included in it.

After the `SA_INIT` exchange, the initiator and responder authenticate each other by signing a portion of the message transcript. Notably the initiator signs:

$$\mathsf{hash}(\underbrace{\mathtt{SA\_INIT}(ck|\mathtt{SA}_i|g^x|n_i|info_i)}_{m_1}|n_r|\mathsf{mac}(k_i, \mathtt{ID}_i))$$

Figure 11 in Appendix depicts an attack on IKEv2 initiator authentication that relies on a transcript collision on this signature. The network attacker intercepts the `SA_INIT` request from I to R and responds with a cookie ck. The initiator I restarts the key exchange by including ck in the new `SA_INIT` request (m_1). However, the attacker has chosen ck in a way that the hash of m_1 is the same as the hash of a tampered `SA_INIT` request m_1' that contains the attacker's Diffie-Hellman public value $g^{x'}$. The attacker sends this tampered request m_1' to the responder and upon receiving a response, it tampers with the response to replace R's Diffie-Hellman key g^y with its own key $g^{y'}$. Note that the attacker does not tamper with the nonces n_i, n_r.

At this point, the attacker knows the shared secrets $g^{x'y}, g^{xy'}$ and encryption keys on the two connections. Moreover the hash used in the signature transcript collides all the way to the $\mathsf{mac}(k_i, \mathtt{ID}_i)$. To complete the attack, the attacker must ensure that k_i is that same at I and R. It can ensure this by choosing x', y' such that $g^{x'y} = g^{xy'}$ (as discussed below). Thereafter, it can forward I's signature to R and hence impersonate I.

Implementing the Attack To implement the attack, we must first find a collision between m_1 and m_1'. We observe that in IKEv2 the length of the cookie is supposed to be at most 64 octets but we found that many implementations allow cookies of up to 2^{16} bytes. We can use this flexibility in computing long collisions.

The attacker finds two length-prefixed bitstrings (C_1, C_2) of L bytes each such that

$$\mathsf{hash}(\mathtt{SA\_INIT}(\underbrace{[C_1|-]}_{ck}|-)) = \mathsf{hash}(\mathtt{SA\_INIT}(\underbrace{C_2}_{ck'}|-))$$

where the length of ck is set to $L+M$, that is, ck has M empty bytes ready to fill in. We set M to the length of the bitstring $\mathtt{SA}_i'|g^{x'}|n_i$ that the attacker wants to send to R in its tampered `SA_INIT` request m_1'. The idea is that the attacker can now stuff the tampered message into ck, and can stuff the original message into $info_i'$ to obtain a transcript collision:

$$\mathsf{hash}(\mathtt{SA\_INIT}(\underbrace{[C_1|\mathtt{SA}_i'|g^{x'}|n_i]}_{ck}|\mathtt{SA}_i|g^x|n_i|info_i)|-) =$$

$$\mathsf{hash}(\mathtt{SA\_INIT}(\underbrace{[C_2]}_{ck}|\mathtt{SA}_i'|g^{x'}|n_i|\underbrace{[\mathtt{SA}_i|g^x|n_i|info_i]}_{info_i'})|-)$$

The collision (C_1, C_2) can be found easily as a chosen-prefix collision attack. Since the collision occurs before any unpredictable value has been included in the message, it can be computed offline; that is, it does not have to be computed while a connection is live. The collision can then be used to break any number of connections between I and R. Such collisions are easy to compute for MD5, but we found that even

though MD5 signatures are allowed by the standard, they are not commonly supported by IKEv2 implementations. However, SHA-1 signatures are mandatory for all IKEv2 implementations, so an offline chosen-prefix collision on SHA-1 is enough to mount the attack. The best known complexity of such collisions is currently 2^{77}, which may be feasible for a powerful adversary (especially if better shortcut attacks on SHA-1 are discovered).

We also observe that the two prefixes are very similar: we only need the length of the cookie to be different. Following the format of IKE message, the length field is on bytes 22 and 23 of the hashed transcript, and all previous bytes must have a fixed value. Hence, we can *almost* use a common-prefix collision attack, if the collision algorithm introduces a difference in bytes 22-23, and no difference in preceding bytes. For MD5, the most efficient collision attacks do not have a compatible message difference, but it seems possible to build a dedicated attack with complexity below 2^{39}. However, for SHA-1, all known collision attacks use differences in every message words, and are thus unsuitable.

The final step to implement the attack is to ensure that $g^{xy'} = g^{x'y}$. To achieve this, we rely on a small subgroup confinement attack. To see a simple example, suppose the attacker chose $x' = y' = 0$; then the two shared secrets would have the value 1. This specific solution would not work in practice because most IKEv2 implementations validate the received Diffie-Hellman public value to ensure that it is larger than 1 and smaller than $p - 1$. However, many IKEv2 implementations support the Diffie-Hellman groups 22-24 that are known to have many small subgroups. These implementations do not validate the incoming public value, and hence are susceptible to similar small subgroup confinement attacks, as discussed in [5]. To complete our transcript collision attack, the MitM can use one such small subgroup to ensure that the shared values on the two connections are the same with high probability.

B. Breaking IKEv1 Initiator Authentication with a Generic Transcript Collision

IKEv1, the predecessor of IKEv2, and is also vulnerable to transcript collision attacks. We briefly outline one attack, without giving more details for lack of space. The initiator's signature in IKEv1 is computed as:

$$\mathsf{sign}(sk_I, \mathsf{prf}(\mathsf{prf}(n_i|n_r, g^{xy}), g^x|g^y|c_i|c_r|\mathtt{SA}_i|\mathtt{ID}_i))$$

A commonly-used PRF function in IKEv1 is HMAC-MD5, and we find a generic transcript collision attack on the outer PRF value that allows initiator impersonation. A man-in-the-middle attacker intercepts a connection between I and R; it tries out many random $g^{y'}$ values on the client-side, and many random values (embedded in) $\mathtt{ID}'_i$ on the server side, until the PRF values on the two sides collide. It can then forward I's signature to R, even though it knows the Diffie-Hellman shared secret. When the PRF is HMAC-MD5, this generic collision attack costs about $2 * 2^{64}$ HMAC computations per connection.

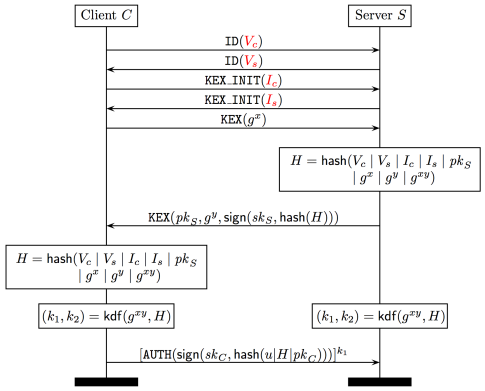


Fig. 8. SSH-2: Key exchange and user authentication.

C. Downgrading SSH-2 to Weak Ciphersuites with a Chosen-Prefix Transcript Collision

Figure 8 depicts the SSH-2 [39] protocol, which implements yet another variation of an authenticated Diffie-Hellman protocol. The client and server exchange identification strings V_c, V_s, negotiate protocol parameters I_c, I_s, and perform a Diffie-Hellman exchange g^x, g^y. To authenticate the exchange, clients and servers sign a session hash, defined as:

$$H = \mathsf{hash}(V_c|V_s|I_c|I_s|pk_S|g^x|g^y|g^{xy})$$

We show that a target collision on this hash value can allow downgrade attaks.

Figure 12 in Appendix depicts a downgrade attack on SSH-2. The network attacker tampers with the key exchange message I_c in one direction and with I_s in the other. It chooses their values in a way such that the following hashes coincide

$$\mathsf{hash}(V_c|V_s|I_c|\underbrace{C_1|-}_{I'_s}) = \mathsf{hash}(V_c|V_s|\underbrace{C_2}_{I'_c})$$

Using this collision, we leave enough space empty in I'_s to stuff the real I_s inside. Consequently the session hashes on the two sides coincide and the connection is completed. In this attack, the MitM does not tamper with the Diffie-Hellman values and hence it does not know the connection keys. However, it manages to

tamper with both I_c and I_s, and can therefore downgrade the negotiate ciphersuite to a weak cryptographic algorithm that the attacker knows how to break.

Implementing the target collision for SSH-2 requires a chosen-prefix attack on SHA-1 which is still considered impractical (at least 2^{77} work). Moreover, since the two tampered fields I_c and I_s are meant to be strings (not bitstrings), we cannot use arbitrary collisions. Still, we find this attack to be an interesting illustration of the use of transcipt collisions for downgrade attacks.

SSH-2 has a peculiar session hash construction, with the shared secret g^{xy} placed at the end. This makes certain kinds of collision attacks more difficult, but we note that this construction is not particularly secure; since it includes the shared secret, the session hash needs to be *non-leaking* in addition to being collision-resistant [4]. Moreover, if the SSH server reuses its Diffie-Hellman public value, this secret suffix becomes vulnerable to key recovery attacks like on APOP [25].

Other variations of SSH allow for more tampering, which may enable new attacks. The SSH Diffie-Hellman Group Exchange protocol [10] allows SSH servers to choose any Diffie-Hellman group for use in the key exchange. So, like in our TLS attacks, a man-in-the-middle attacker can send a bogus or weak group to the client, and use it to control more fields in the session hash and mount new transcript collision attacks.

VII. SLOTH: Responsible Disclosure and Impact

Table I summarizes the attacks discussed in this paper. Three of our attacks on TLS are already practical; others are within the reach of powerful adversaries.

Our attacks on TLS were publicly disclosed under the acronym SLOTH (security losses from obsolete and truncated transcript hashes) and were assigned a protocol-level CVE-2015-7575. We informed the authors of affected protocol specifications and developers for various TLS libraries. We recommended that protocols and implementations should stop using MD5-based signatures and other weak hash constructions. Our disclosure and recommendations resulted in the following security updates:

1) TLS 1.3 draft 7 stopped truncating the Finished MACs and started using the full HMAC output.
2) TLS 1.3 draft 8 deprecated MD5 signatures.
3) The Token Binding Protocol draft 2 removed `tls-unique` and moved to a stronger channel binding.
4) Akamai servers disabled support or RSA-MD5 client and server signatures.
5) Red Hat issued backported patches RHEL 6 and 7 to disable MD5 signatures in their version of OpenSSL version 1.0.1e.
6) NSS 3.21 (FireFox 43) disabled support for MD5 server signatures; MD5-based client signatures were already disabled.
7) GnuTLS 3.3.15 disabled MD5 signatures in the default configuration.
8) BouncyCastle Java 1.54 (C# 1.8.1) disabled MD5 signatures in the default configuration.
9) Oracle and IBM are updating the TLS implementation in their Java runtimes to disable MD5 signatures in the default configuration.
10) mbedTLS is being updated to disable MD5 server signatures; MD5 client signatures were already disabled.

These changes impact the Firefox and Android browsers, about 31% of web servers, most Java application servers and their clients, and many other custom applications that use less well-known TLS libraries. We are maintaining a website with the currently known attacks, affected software, and disclosure status at our website:

http://sloth-attack.org .

VIII. Conclusions

We have demonstrated that the use of MD5 and truncated HMACs for authenticating transcripts in various Internet protocols leads to exploitable chosen-prefix and generic collision attacks. We also showed several unsafe uses of SHA-1 that will become dangerous when more efficient collision-finding algorithms for SHA-1 are discovered. In all cases, the complexity of our transcript collision attacks are significantly lower than the estimated work for a second preimage attack on the underlying hash function. This definitively settles the debate on whether the security of mainstream cryptographic protocols depend on collision resistance. The answer is *yes*, cryptographers were right. Except in rare cases, mainstream protocols do require collision resistance for protection against man-in-the-middle transcript collision attacks. Consequently, we strongly recommend that weak hash functions like MD5 and SHA-1 should not just be deprecated; they should be forcefully disabled in existing protocols.

An open research question is whether it is possible to design key exchange protocols that will be resilient to new collision attacks. One strategy is to use a commitment scheme (like ZRTP [40]) that would make it more difficult for a man-in-the-middle to tamper with the transcript. However, such schemes may still be vulnerable to certain shortcut collisions [19]. For signatures, randomized hashing [13] provides a different way forward but its integration into a complex protocol like TLS would need to be carefully analyzed.

Protocol	Property	Mechanism	Attack	Collision Type	Precomp.	Work/conn.	Preimage	Wall-clock time
TLS 1.2	Client Auth	RSA-MD5	Impersonation	Chosen Prefix		2^{39}	2^{128}	48 core hours
TLS 1.3	Server Auth	RSA-MD5	Impersonation	Chosen Prefix		2^{39}	2^{128}	48 core hours
TLS 1.0-1.2	Channel Binding	HMAC (96 bits)	Impersonation	Generic		2^{48}	2^{96}	80 GPU days
TLS 1.2	Server Auth	RSA-MD5	Impersonation	Generic	2^X conn.	2^{128-X}	2^{128}	
TLS 1.0-1.1	Handshake Integrity	MD5 | SHA-1	Downgrade	Chosen Prefix		2^{77}	2^{160}	
IKE v1	Initiator Auth	HMAC-MD5	Impersonation	Generic		2^{65}	2^{128}	
IKE v2	Initiator Auth	RSA-SHA-1	Impersonation	Chosen Prefix	2^{77}	0	2^{160}	
SSH-2	Exchange Integrity	SHA-1	Downgrade	Chosen Prefix		2^{77}	2^{160}	

TABLE I. SUMMARY OF TRANSCRIPT COLLISION ATTACKS ON INTERNET PROTOCOLS

REFERENCES

[1] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, , and P. Zimmermann. Imperfect forward secrecy: How diffie-hellman fails in practice. In *ACM CCS*, 2015.

[2] J. Altman, N. Williams, and L. Zhu. Channel bindings for TLS. IETF RFC 5929, 2010.

[3] S. Bellovin and E. Rescorla. Deploying a new hash algorithm. In *NDSS*, 2006.

[4] F. Bergsma, B. Dowling, F. Kohlar, J. Schwenk, and D. Stebila. Multi-ciphersuite security of the secure shell (ssh) protocol. In *ACM CCS*, pages 369–381, 2014.

[5] K. Bhargavan, A. Delignat-Lavaud, and A. Pironti. Verified contributive channel bindings for compound authentication. In *NDSS*, 2015.

[6] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti, and P.-Y. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE S&P (Oakland)*, 2014.

[7] I. B. Damgård. A design principle for hash functions. In *CRYPTO'89*, 1990.

[8] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. IETF RFC 5246, 2008.

[9] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. Internet Draft, 2014.

[10] M. Friedl, N. Provos, and W. Simpson. Diffie-Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol. IETF RFC 4419, 2006.

[11] F. Giesen, F. Kohlar, and D. Stebila. On the security of TLS renegotiation. In *ACM CCS*, 2013.

[12] D. Gillmor. Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for TLS. Internet Draft, 2015.

[13] S. Halevi and H. Krawczyk. Strengthening digital signatures via randomized hashing. In *CRYPTO*, 2006.

[14] B. Hill, D. Baghdasaryan, B. Blanke, R. Lindemann, and J. Hodges. FIDO UAF Application API and Transport Binding Specification v1.0. Draft Specification, 2015.

[15] P. Hoffman. Use of Hash Algorithms in Internet Key Exchange (IKE) and IPsec. IETF RFC 4894, 2007.

[16] P. Hoffman and B. Schneier. Attacks on Cryptographic Hashes in Internet Protocols. IETF RFC 4270, 2005.

[17] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. On the security of TLS-DHE in the standard model. In *CRYPTO*, 2012.

[18] A. Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In *CRYPTO*, 2004.

[19] J. Kelsey and T. Kohno. Herding hash functions and the nostradamus attack. In *EUROCRYPT*, 2006.

[20] D. Knuth. Seminumerical algorithms, volume 2 of the art of computer programming, 1981.

[21] H. Krawczyk. SIGMA: The SIGn-and-MAc approach to authenticated Diffie-Hellman and its use in the IKE protocols. In *CRYPTO*. 2003.

[22] H. Krawczyk, K. G. Paterson, and H. Wee. On the security of the TLS protocol: A systematic analysis. In *CRYPTO*, 2013.

[23] A. Langley. Transport Layer Security (TLS) Snap Start. Internet Draft, 2010.

[24] A. Langley. Transport Layer Security (TLS) Next Protocol Negotiation Extension. Internet Draft, 2012.

[25] G. Leurent. Practical key-recovery attack against APOP, an MD5-based challenge-response authentication. *IJACT*, 1(1):32–46, 2008.

[26] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel. A cross-protocol attack on the TLS protocol. In *ACM CCS*, 2012.

[27] F. Mendel, C. Rechberger, and M. Schläffer. MD5 is weaker than weak: Attacks on concatenated combiners. In *ASIACRYPT*, 2009.

[28] A. Menon-Sen, N. Williams, A. Melnikov, and C. Newman. Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms. IETF RFC 5802, 2010.

[29] R. C. Merkle. A certified digital signature. In *CRYPTO'89*, 1990.

[30] J. M. Pollard. A monte carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[31] J. M. Pollard. Monte carlo methods for index computation. *Mathematics of computation*, 32(143):918–924, 1978.

[32] A. Popov, M. Nystroem, D. Balfanz, and A. Langley. The Token Binding Protocol Version 1.0. Internet Draft, 2015.

[33] M. Stevens. Hashclash. https://marc-stevens.nl/p/hashclash/.

[34] M. Stevens. Counter-cryptanalysis. In *CRYPTO*, 2013.

[35] M. Stevens. New collision attacks on SHA-1 based on optimal joint local-collision analysis. In *EUROCRYPT*, 2013.

[36] M. Stevens, A. K. Lenstra, and B. de Weger. Chosen-prefix collisions for MD5 and applications. *IJACT*, 2(4):322–359, 2012.

[37] P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptology*, 12(1):1–28, 1999.

[38] X. Wang and H. Yu. How to break MD5 and other hash functions. In *EUROCRYPT*, 2005.

[39] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253 (Proposed Standard), 2006.

[40] P. Zimmermann. ZRTP: Media Path Key Agreement for Unicast Secure RTP. IETF RFC 6189, 2012.

H	Collision	CPC
Generic	$2^{N/2}$	$2^{N/2}$
$H_1 \mid H_2$	$2^{N_1/2}N_2/2 + 2^{N_2/2}$	$2^{N_1/2}N_2/2 + 2^{N_2/2}$
MD5	2^{16}	2^{39}
SHA-1	2^{61}	2^{77}
MD5 | SHA-1	2^{67}	2^{77}

TABLE II. COMPLEXITY OF FINDING COLLISIONS IN VARIOUS HASH CONSTRUCTIONS

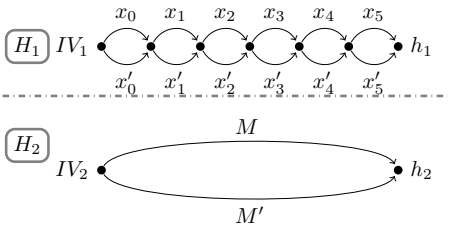


Fig. 9. Multi-collision attack

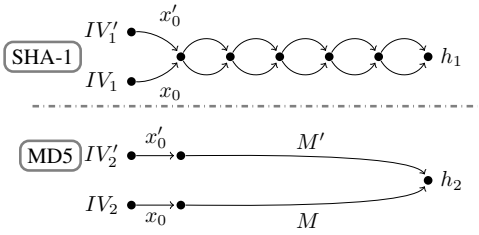


Fig. 10. CPC attack against MD5 | SHA-1

APPENDIX

A. Attacks against Hash Functions

We now give more details about attacks against hash function, considering both generic attacks and dedicated attacks against widely-used functions MD5 and SHA-1. The main results are summarizex in Table II.

Generic collision attacks While a basic collision attack requires to compute and store $2^{N/2}$ images of the hash function, it is possible to mount a parallel and memory-less attack with a very small overhead. The main idea was introduced by Pollard as the Rho algorithm for factorization [30] and discrete logarithms [31], and was later generalized to collision search. The hash function is first restricted from $\{0,1\}^* \to \{0,1\}^N$ to $\{0,1\}^N \to \{0,1\}^N$, so that it can be iterated. After some number of steps, a chain of iterations reaches a cycle, and the graph will have the shape of the greek letter ρ. On average, the cycle has length $O(2^{N/2})$ and is reached after $O(2^{N/2})$ steps. The point where the tail of the meets with the cycle reveals a collision in the hash function. It can be detected in time $O(2^{N/2})$ with little or no memory, using various cycle detection methods, such as Floyd's algorithm [20] (also known as tortoise and hare).

Some variants of this attack using distinguished points can be parallelized efficiently. We now describe a parallel version of Pollard's Lambda algorithm, as described by van Oorschot and Wiener [37], using c CPUs. Each CPU will compute iteration chains of the function H, and stop when reaching a *distinguished point*, that is a point with some easy to test property. For instance, we stop a computation when the ending point satisfies $x < 2^{N/2}\alpha c$ for some small constant α, so that the expected length of a chain is $2^{N/2}/\alpha c$. When a chain is finished, we store the starting point, the length, and the ending point. We generate αc chains in this way, so that the function has been evaluated about $2^{N/2}$ times, and there is a high probability that there was a collision. The important idea of this attack is that if a given point is reached by two different chains, both chain will stop at the same distinguished point. Therefore, we look at the ending points of the chains, and when a collision is detected, we restart the chains from the starting point in order to locate the collision. This attack requires about $2^{N/2}$ evaluations of H, and a memory of αc when using c CPUs.

This attack can be tweaked for a chosen-prefix collision attack using an auxiliary function $g\colon \{0,1\}^N \to \{0,1\}^N$ defined as:

$$g(x) = \begin{cases} H(P|x) & \text{if } x \text{ is even} \\ H(P'|x) & \text{if } x \text{ is odd.} \end{cases}$$

Collisions in g can be found with the previous techniques. With probability $1/2$ a collision is g is between an even x and an odd x' (or vice versa), this implies a chosen-prefix collision $H(P|x) = H(P'|x')$. An accurate complexity analysis is provided in [37]: a collision is expected to be found after $\sqrt{\pi 2^N/2}$ evaluations. For a chosen-prefix collision, we expect to find two collisions in g after $\sqrt{\pi 2^N}$ evaluations.

Concatenation Collisions in the concatenation of two hash functions $H_1|H_2$ can be found with roughly the same effort as breaking the strongest one of the two, using the multi-collision technique of Joux [18].

The adversary first finds a collision pair (x_0, x'_0) for H_1, starting from the initialization value of H_1. Then it finds a collision pair (x_1, x'_1) starting from $H_1(x_0) = H_1(x'_0)$. This defines 4 messages with the same H_1-digest: $x_0|x_1$, $x_0|x'_1$, $x'_0|x_1$, $x'_0|x'_1$. After $N_2/2$ steps, this defines a set of $2^{N_2/2}$ messages with the

same H_1-digest. With high probability, two of these messages have the same H_2-digest as well (see Figure 9). Therefore, one can find a collision in $H_1|H_2$ with a complexity only $N_2/2 \times 2^{N_1/2} + 2^{N_2/2}$. For MD5 | SHA-1, this translates to 2^{80}, roughly as much as a generic collision attack on SHA-1.

Better attacks against MD5 | SHA-1 result from the combination of Joux's multicollision technique with shortcut attacks against SHA-1. A collision attack can be build for a cost of $64 \times 2^{61} + 2^{64} \approx 2^{67}$ (building sequentially 64 collisions for MD5). For a chosen-prefix collision, we first perform a chosen-prefix collision against SHA-1, to generate messages (x, x') such that SHA-1$(P|x)$ = SHA-1$(P'|x')$. Then we build a multicollision in SHA-1 starting from this value, and we evaluate MD5 over a set of 2^{64} messages in order to find a collision. The total cost is about $2^{77} + 64 \times 2^{61} + 2^{64} \approx 2^{77}$ (see Figure 10).

Moreover it has been shown that it is possible to combine cryptanalytic shortcuts both on SHA-1 and MD5, assuming that collision attacks against SHA-1 improve in the future [27]. This may allow collision attacks against MD5 | SHA-1 with less than 2^{64} work. Table II summarizes the currently-known complexities for computing various hash collisions.

B. Transcript Collision Attacks on IKEv2 and SSH-2

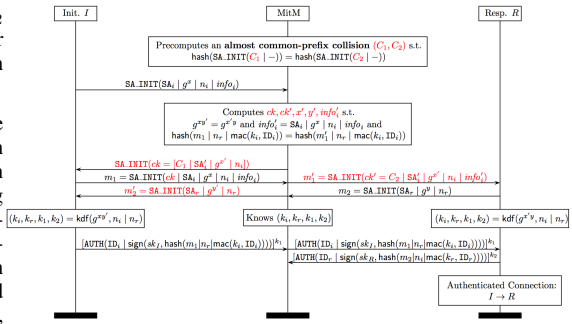


Fig. 11. Man-in-the-middle initiator impersonation attack on IKEv2. The initiator I connects to the responder R but a man-in-the-middle attacker A intercepts and tampers with some messages (shown in red). A pecomputes a collision (C_1, C_2) between the prefixes of two SA_INIT messages that both begin with a cookie payload. Then by sending a carefully crafted cookie to I, A can trigger a transcript collision on the initiator signature, which it can then forward to R, thereby impersonating I on a connection that it controls.

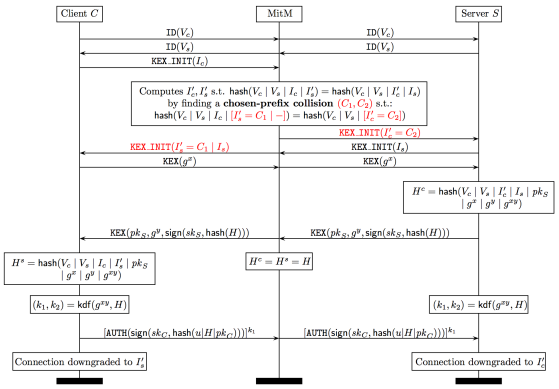


Fig. 12. Man-in-the-middle downgrade attack on SSH-2. The client C connects to a server S, but a network attacker A tampers with the key exchange messages (shown in red) to downgrade them to a weak ciphersuite. To succeed, A must compute a chosen-prefix collision on the session hash H after receiving C's key exchange message.

New Generic Attacks Against Hash-based MACs

Gaëtan Leurent[1], Thomas Peyrin[2], and Lei Wang[2]

[1] Université Catholique de Louvain, Belgium
gaetan.leurent@uclouvain.be
[2] Nanyang Technological University, Singapore
thomas.peyrin@gmail.com wang.lei@ntu.edu.sg

Abstract. In this paper we study the security of hash-based MAC algorithms (such as HMAC and NMAC) above the birthday bound. Up to the birthday bound, HMAC and NMAC are proven to be secure under reasonable assumptions on the hash function. On the other hand, if an n-bit MAC is built from a hash function with a l-bit state ($l \geq n$), there is a well-known existential forgery attack with complexity $2^{l/2}$. However, the remaining security after $2^{l/2}$ computations is not well understood. In particular it is widely assumed that if the underlying hash function is sound, then a generic universal forgery attack should require 2^n computations and some distinguishing (*e.g.* distinguishing-H but not distinguishing-R) and state-recovery attacks should also require 2^l computations (or 2^k if $k < l$).

In this work, we show that above the birthday bound, hash-based MACs offer significantly less security than previously believed. Our main result is a generic distinguishing-H and state-recovery attack against hash-based MACs with a complexity of only $\tilde{O}(2^{l/2})$. In addition, we show a key-recovery attack with complexity $\tilde{O}(2^{3l/4})$ against HMAC used with a hash functions with an internal checksum, such as GOST. This surprising result shows that the use of a checksum might actually weaken a hash function when used in a MAC. We stress that our attacks are generic, and they are in fact more efficient than some previous attacks proposed on MACs instanciated with concrete hash functions.

We use techniques similar to the cycle-detection technique proposed by Peyrin *et al.* at Asiacrypt 2012 to attack HMAC in the related-key model. However, our attacks works in the single-key model for both HMAC and NMAC, and without restriction on the key size.

Key words: NMAC, HMAC, hash function, distinguishing-H, key recovery, GOST.

1 Introduction

Message Authentication Codes (MACs) are crucial components in many security systems. A MAC is a function that takes a k-bit secret key K and an arbitrarily long message M as inputs, and outputs a fixed-length tag of size n bits. The tag is used to authenticate the message, and will be verified by the receiving party using the same key K. Common MAC algorithms are built from block ciphers (*e.g.* CBC-MAC), from hash functions (*e.g.* HMAC), or from universal

hash functions (*e.g.* UMAC). In this paper we study MAC algorithms based on hash functions.

As a cryptographic primitive, a MAC algorithm should meet some security requirements. It should be impossible to recover the secret key except by exhaustive search, and it should be computationally impossible to forge a valid MAC without knowing the secret key, the message being chosen by the attacker (existential forgery) or given as a challenge (universal forgery). In addition, cryptanalysts have also studied security notions based on distinguishing games. Informally, the distinguishing-R game is to distinguish a MAC construction from a random function, while the distinguishing-H game is to distinguish a known MAC construction (*e.g.* `HMAC`) instantiated with a known component (*e.g.* `SHA-1`) under a random key from the same construction instantiated with a random component (*e.g.* `HMAC` with a fixed input length random function).

One of the best known MAC algorithm is `HMAC` [2], designed by Bellare *et al.* in 1996. `HMAC` is now widely standardized (by ANSI, IETF, ISO and NIST), and widely deployed, in particular for banking processes or Internet protocols (*e.g.* SSL, TLS, SSH, IPSec). It is a single-key version of the `NMAC` construction, the latter being built upon a keyed iterative hash function H_K, while `HMAC` uses an unkeyed iterative hash function H:

$$\texttt{NMAC}(K_{out}, K_{in}, M) = H_{K_{out}}(H_{K_{in}}(M))$$
$$\texttt{HMAC}(K, M) = H(K \oplus \texttt{opad} \parallel H(K \oplus \texttt{ipad} \parallel M))$$

where `opad` and `ipad` are predetermined constants, and where K_{in} denotes the inner key and K_{out} the outer one.

More generally, a MAC algorithm based on a hash function uses the key at the beginning and/or at the end of the computation, and updates an l-bit state with a compression function. The security of MAC algorithms is an important topic, and both positive and negative results are known. On the one hand, there is a generic attack with complexity $2^{l/2}$ based on internal collisions and length extension [17]. This gives an existential forgery attack, and a distinguishing-H attack. One the other hand, we have security proofs for several MAC algorithms such as HMAC and sandwich-MAC [2,1,26]. Roughly speaking, the proofs show that some MAC algorithms are secure up to the birthday bound ($2^{l/2}$) under various assumptions on the compression function and hash function.

Thanks to those results, one may consider that the security of hash-based MAC algorithms is well understood. However, there is still a strong interest in the security above the birthday bound. In particular, it is very common to expect security 2^k for key recovery attacks if the hash function is sound; the Encyclopedia of Cryptography and Security article on `HMAC` states explicitly [16] "A generic key recovery attack requires $2^{n/2}$ known text-MAC pairs and 2^{n+1} time" (assuming $n = l = k$). Indeed, key recovery attacks against `HMAC` with a concrete hash function with complexity between $2^{l/2}$ and 2^l have been considered as important results [9,24,27]. Similarly, the best known dinstinguishing-H and state-recovery attacks have a complexity of 2^l (or 2^k if $k < l$), and distinguishing-

Table 1. Comparison of our generic attacks on HMAC, and some previous attacks on concrete hash function. We measure the complexity as the number of calls to the compression function (*i.e.* number of messages times message length)

Function	Attack	Complexity	M. len	Notes	Ref
HMAC-MD5	dist-H, state rec.	2^{97}	2		[25]
HMAC-SHA-0	dist-H	2^{100}	2		[12]
HMAC-HAVAL (3-pass)	dist-H	2^{228}	2		[12]
HMAC-SHA-1 (43 first steps)	dist-H	2^{154}	2		[12]
HMAC-SHA-1 (58 first steps)	dist-H	2^{158}	2		[18]
HMAC-SHA-1 (61 mid. steps)	dist-H	2^{100}	2		[18]
HMAC-SHA-1 (62 mid. steps)	dist-H	2^{157}	2		[18]
Generic attacks:					
hash-based MAC (*e.g.* HMAC)	dist-H	$O(2^{l/2})$	$2^{l/2}$		Sec. 4
	state rec.	$\tilde{O}(2^{l/2})$	$2^{l/2}$		Sec. 5
	dist-H, state rec.	$O(2^{l-s})$	2^s	$s \leq l/4$	full version
HMAC with a checksum	key rec.	$O(l \cdot 2^{3l/4})$	$2^{l/2}$		Sec. 7
		$O(l \cdot 2^{3l/4})$	$2^{l/4}$		full version
HMAC-MD5*	dist-H, state rec.	2^{67}, 2^{78}	2^{64}		Sec. 4, 5
		$O(2^{96})$	2^{32}		full version
HMAC-HAVAL† (any)	dist-H, state rec.	$O(2^{202})$	2^{54}		full version
HMAC-SHA-1†	dist-H, state rec.	$O(2^{120})$	2^{40}		full version
HMAC-GOST*	key rec.	2^{200}	2^{128}		Sec. 7
		2^{200}	2^{64}		full version

* The MD5 and GOST specifications allow arbitrary-length messages
† The SHA-1 and HAVAL specifications limits the message length to 2^{64} bits (and 2^{64} bits is 2^{54} blocks)

H attacks on HMAC with a concrete hash function with complexity between $2^{l/2}$ and 2^l have been considered as important results [12,18,25].

Our contributions. In this paper we revisit the security of hash-based MAC above the birthday bound. We describe a generic distinguishing-H attack in the single key model and with complexity of only $O(2^{l/2})$ computations, thus putting an end to the long time belief of the cryptography community that the best generic distinguishing-H on NMAC and HMAC requires $\Omega(2^l)$ operations. Instead, we show that a distinguishing-H attack is not harder than a distinguishing-R attack. Our results actually invalidate some of the recently published cryptanalysis works on HMAC when instantiated with real hash functions [12,18,25]

Our method is based on a cycle length detection, like the work of Peyrin *et. al* [14], but our utilization is quite different and much less restrictive: instead of iterating many times HMAC with small messages and a special related-key, we will use only a few iterations with very long messages composed of almost the same message blocks to observe the cycle and deduct information from it. Overall,

unlike in [14], our technique works in the single-key model, for both HMAC and NMAC, and can be applied for any key size. In addition, leveraging our new ideas, we provide a single-key internal state recovery for any hash-based MAC with only $O(l \cdot \log(l) \cdot 2^{l/2})$ computations.

We also introduce a different approach to reduce the length of the queried messages for distinguishing-H and internal-state-recovery attacks. Due to the limited space, we only give a quick overview of these shorter-message attacks, and refer the reader to the full version of this paper for more details.

Finally, this internal state recovery can be transformed into a single-key key recovery attack on HMAC with complexity $O(l \cdot 2^{3l/4})$ when instantiated with a hash function using a checksum, such as the GOST hash function [6]. A surprising corollary to our results is that incorporating a checksum to a hash function seems to actually reduce the security of the overall HMAC design.

We give an overview of our results, and a comparison with some previous analysis in Table 1.

The description of HMAC/NMAC algorithms and their security are given in Section 2 and we recall in Section 3 the cycle-detection ideas from [14]. Then, we provide in Section 4 the generic distinguishing-H attack. We give in Sections 5 and 6 internal state recovery methods, and finally describe our results when the hash function incorporates a checksum in Section 7.

2 Hash-based MAC algorithms

In this paper, we study a category of MAC algorithms based on hash functions, where the key is used at the beginning and at the end of the computation, as described in Figure 1. More precisely, we consider algorithms where: the message processing is done by updating an internal state x using a compression function h; the state is initialized with a key dependent value I_k; and the tag is computed from the last state x_p and the key K by an output function g.

$$x_0 = I_K \qquad x_{i+1} = h(x_i, m_i) \qquad \mathtt{MAC}_K(M) = g(K, x_p, |M|)$$

In particular, this description covers NMAC/HMAC [2], envelope-MAC [21], and sandwich-MAC [26]. The results described in Sections 4 and 5 can be applied to any hash-based MAC, but we focus on HMAC for our explanations because it is the most widely used hash-based MAC, and its security has been widely analyzed already. On the other hand the result of Section 7 is specific to MAC algorithms that process the key as part of the message, such as HMAC.

2.1 Description of NMAC and HMAC

A hash function H is a function that takes an arbitrary length input message M and outputs a fixed hash value of size n bits.

Virtually every hash function in use today follows an iterated structure like the classical Merkle-Damgård construction [13,4]. Iterative hash functions are

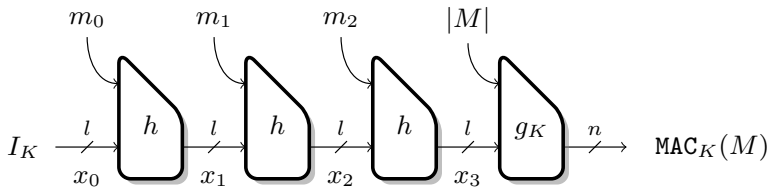


Fig. 1. Hash-based MAC. Only the initial value and the final transformation are keyed.

built upon successive applications of a so-called compression function h, that takes a b-bit message block and a l-bit chaining value as inputs and outputs a l-bit value (where $l \geq n$). An output function can be included to derive the n-bit hash output from the last chaining variable and from the message length. When $l > n$ (resp. when $l = n$) we say that the hash function is wide pipe (resp. narrow pipe).

The message M is first padded and then divided into blocks m_i of b bits each. Then, the message blocks are successively used to update the l-bit internal state x_i with the compression function h. Once all the message blocks have been processed, the output function g is applied to the last internal state value x_p.

$$x_0 = IV \qquad x_{i+1} = h(x_i, m_i) \qquad \mathsf{hash} = g(x_p, |M|)$$

The MAC algorithm `NMAC` [2] uses two l-bit keys K_{out} and K_{in}. `NMAC` replaces the public IV of a hash function H by a secret key K to produce a keyed hash function $H_K(M)$. `NMAC` is then defined as:

$$\mathtt{NMAC}(K_{out}, K_{in}, M) = H_{K_{out}}(H_{K_{in}}(M)).$$

The MAC algorithm `HMAC` [2] is a single-key version of `NMAC`, with $K_{out} = h(IV, K \oplus \mathtt{opad})$ and $K_{in} = h(IV, K \oplus \mathtt{ipad})$, where `opad` and `ipad` are b-bit constants. However, a very interesting property of `HMAC` for practical utilization is that it can use any key size and can be instantiated with an unkeyed hash function (like `MD5`, `SHA-1`, etc. which have a fixed IV):

$$\mathtt{HMAC}(K, M) = H(K \oplus \mathtt{opad} \parallel H(K \oplus \mathtt{ipad} \parallel M)).$$

where $\parallel$ denotes the concatenation operation. For simplicity of the description and without loss of generality concerning our attacks, in the rest of this article we assume that the key can fit in one compression function message block, i.e. $k \leq b$ (note that K is actually padded to b bits if $k < b$).

2.2 Security of `NMAC` and `HMAC`

In [1], Bellare proved that the `NMAC` construction is a pseudo-random function (PRF) under the sole assumption that the internal compression function h (keyed

by the chaining variable input) is a PRF. The result can be transposed to HMAC as well under the extra assumption that h is also a PRF when keyed by the message input.

Concerning key recovery attacks, an adversary should not be able to recover the key in less than 2^k computations for both HMAC and NMAC. In the case of NMAC one can attack the two keys K_{out} and K_{in} independently by first finding an internal collision and then using this colliding pair information to brute force the first key K_{in}, and finally the second one K_{out}.

Universal or existential forgery attacks should cost 2^n computations for a perfectly secure n-bit MAC. However, the iterated nature of the hash functions used inside NMAC or HMAC allows a simple existential forgery attack requiring only $2^{l/2}$ computations [17]. Indeed, with $2^{l/2}$ queries, an adversary can first find an internal collision between two messages (M, M') of the same length during the first hash call. Then, any extra block m added to both of these two messages will lead again to an internal collision. Thus, the attacker can simply forge a valid MAC by only querying for $M \parallel m$ and deducing that $M' \parallel m$ will have the same MAC value.

Concerning distinguishers on HMAC or NMAC, two types have been discussed in the literature: Distinguishing-R and Distinguishing-H attacks, defined below:

Distinguishing-R. Let $\mathcal{F}_n$ be the set of n-bit output functions. We denote F_K the oracle on which the adversary $\mathcal{A}$ can make queries. The oracle is instantiated either with $F_K = \texttt{HMAC}_K$ (with K being a randomly chosen k-bit key) or with a randomly chosen function R_K from $\mathcal{F}_n$. The goal of the adversary is to distinguish between the two cases and its advantage is given by

$$Adv(\mathcal{A}) = |\Pr[\mathcal{A}(\texttt{HMAC}_K) = 1] - \Pr[\mathcal{A}(R_K) = 1]| \,.$$

Obviously the collision-based forgery attack detailed above gives directly a distinguishing-R attack on NMAC and HMAC. Thus, the expected security of HMAC and NMAC against distinguishing-R attacks is $2^{l/2}$ computations.

Distinguishing-H. The attacker is given access to an oracle $\texttt{HMAC}_K$ and the compression function of the HMAC oracle is instantiated either with a known dedicated compression function h or with a random chosen function r from $\mathcal{F}_l^{b+l}$ (the set of $(b+l)$-bit to l-bit functions), which we denote $\texttt{HMAC}_K^h$ and $\texttt{HMAC}_K^r$ respectively. The goal of the adversary is to distinguish between the two cases and its advantage is given by

$$Adv(\mathcal{A}) = \left|\Pr[\mathcal{A}(\texttt{HMAC}_K^h) = 1] - \Pr[\mathcal{A}(\texttt{HMAC}_K^r) = 1]\right| .$$

The distinguishing-H notion was introduced by Kim *et al.* [12] for situations where the attacker wants to check which cryptographic hash function is embedded in HMAC. To the best of our knowledge, the best known generic distinguishing-H attack requires 2^l computations.

Related-key attacks. At Asiacrypt 2012 [14], a new type of generic distinguishing (distinguishing-R or distinguishing-H) and forgery attacks for HMAC was proposed. These attacks are in the related-key model and can apply even to wide-pipe proposals, but they only work for HMAC, and only when a special restrictive criterion is verified: the attacker must be able to force a specific difference between the inner and the outer keys (with the predefined values of opad and ipad in HMAC, this criterion is verified when $k = b$). The idea is to compare the cycle length when iterating the HMAC construction on small messages with a key K, and the cycle length when iterating with a key $K' = K \oplus \texttt{opad} \oplus \texttt{ipad}$.

Attacks on instantiations with concrete hash functions. Because of its widespread use in many security applications, HMAC has also been carefully scrutinized when instantiated with a concrete hash function, exploiting weaknesses of some existing hash function. In parallel to the recent impressive advances on hash function cryptanalysis, the community analyzed the possible impact on the security of HMAC when instantiated with standards such as MD4 [19], MD5 [20], SHA-1 [22] or HAVAL. In particular, key-recovery attacks have been found on HMAC-MD4 [9,24] and HMAC-HAVAL [27]. Concerning the distinguishing-H notion, one can cite for example the works from Kim *et al.* [12], Rechberger *et al.* [18] and Wang *et al.* [25].

However, to put these attacks in perspective, it is important to know the complexity of *generic* attacks, that work even with a good hash function.

3 Cycle detection for HMAC

Our new attacks are based on some well-known properties of random functions.

3.1 Random mapping properties on a finite set

Let us consider a random function f mapping n bits to n bits and we denote $N = 2^n$. We would like to know the structure of the functional graph defined by the successive iteration of this function, for example the expected number of components, cycles, etc. First, it is easy to see that each component will contain a single cycle with several trees linked to it. This has already been studied for a long time, and in particular we recall two theorems from Flajolet and Odlyzko [7].

Theorem 1 ([7, Th. 2]). *The expectations of the number of components, number of cyclic points, number of terminal points, number of image points, and number of k-th iterate image points in a random mapping of size N have the asymptotic forms, as $N \to \infty$:*

(i) *# Components:* $\frac{1}{2}\log N$
(ii) *# Cyclic nodes:* $\sqrt{\pi N/2}$
(iii) *# Terminal nodes:* $e^{-1}N$
(iv) *# Image points:* $(1 - e^{-1})N$
(v) *# k-th iterate images:* $(1 - \tau_k)N$, *with* $\tau_0 = 0$, $\tau_{k+1} = e^{-1+\tau_k}$.

In particular, a random mapping has only a logarithmic number of distinct components, and the number of cyclic points follows the square root of N.

By choosing a random starting point P and iterating the function f, one will follow a path in the functional graph starting from P, that will eventually connect to the cycle of the component in which P belongs, and we call **tail length** the number of points in this path. Similarly, we call **cycle length** the number of nodes in the cycle. Finally, the number of points in the non-repeating trajectory from P is called the **rho length**, and we call **α-node** of the path the node that connects the tail and the cycle.

Theorem 2 ([7, Th. 3]). *Seen from a random point in a random mapping of size N, the expectations of the tail length, cycle length, rho length, tree size, component size, and predecessors size have the following asymptotic forms:*

(i) Tail length (λ): $\sqrt{\pi N/8}$
(ii) Cycle length (μ): $\sqrt{\pi N/8}$
(iii) Rho length ($\rho = \lambda + \mu$): $\sqrt{\pi N/2}$
(iv) Tree size: $N/3$
(v) Component size: $2N/3$
(vi) Predecessors size: $\sqrt{\pi N/8}$

One can see that, surprisingly, in a random mapping most of the points tend to be grouped together in a single giant component, and there is a giant tree with a significant proportion of the points. The asymptotic expectation of the maximal features is given by Flajolet and Sedgewick [8].

Theorem 3 ([8, VII.14]). *In a random mapping of size N, the largest tree has an expected size of $\delta_1 N$ with $\delta_1 \approx 0.48$ and the largest component has an expected size of $\delta_2 N$ with $\delta_2 \approx 0.7582$.*

These statistical properties will be useful to understand the advantage of our attacks. We show the functional graph of a simple random-looking function in Figure 4 in the Appendix.

3.2 Using cycle-detection to obtain some secret information

In this article and as in [14], we will study the functional graph structure of a function to derive a distinguisher or obtain some secret information. More precisely, in [14] Peyrin *et al.* observed that the functional graph structure of `HMAC` was the same when instantiated with a key K or with a related key $K' = K \oplus \texttt{ipad} \oplus \texttt{opad}$ (note that in order to be able to query this related-key K', the key K has to be of size b or $b-1$, which is quite restrictive). This is a property that should not exist for a randomly chosen function and they were able to detect this cycle structure by measuring the cycle length in both cases K and K', and therefore obtaining a distinguishing-R attack for `HMAC` in the related-key model. In practice, the attacker can build and observe the functional graph of `HMAC` by simply successively querying the previous n-bit output as new message input.

In this work, instead of studying the structure of the functional graph of `HMAC` directly, we will instead study the functional graph of the internal compression

function h with a fixed message block: we denote $h_M(X) = h(X, M)$. We aim to obtain some information on h_M that we can propagate outside the HMAC structure. This is therefore perfectly suited for a distinguishing-H attack, which requires the attacker to exhibit a property of h when embedded inside the HMAC construction. We can traverse the functional graph of h_M by querying the HMAC oracle with a long message composed of many repetitions of the fixed message block M. The issue is now to detect some properties of the functional graph of h_M inside HMAC and without knowing the secret key. We explain how to do that in the next section.

4 Distinguishing-H attack for hash-based MACs

In the rest of the article, we use the notation $[x]^k$ to represent the successive concatenation of k message blocks x, with $[x] = [x]^1$.

4.1 General description

In order to derive a distinguishing-H attack, we need to do some offline computations with the target compression function h and use this information to compare online with the function embedded in the MAC oracle. We use the structure of the functional graph of $h_{[0]}$ to derive our attack (of course we can choose any fixed message block). We can travel in the graph by querying the oracle using consecutive [0] message blocks. However, since the key is unknown, we do not know where we start or where we stop in this graph. We have seen in the previous section that the functional graph of a random function is likely to have a giant component containing most of the nodes. We found that the cycle size of the giant component of $h_{[0]}$ is a property that can be efficiently tested.

More precisely, we first compute the cycle size of the giant component of $h_{[0]}$ offline; we denote it as L. Then, we measure the cycle size of the giant component of the unknown function by querying the MAC oracle with long messages composed of many consecutive [0] message blocks. If no length padding is used in the hash function, this is quite simple: we just compare $\texttt{MAC}([0]^{2^{l/2}})$ and $\texttt{MAC}([0]^{2^{l/2}+L})$. With a good probability, the sequence of $2^{l/2}$ zero block is sufficiently long to enter the cycle, and if the cycle has length L, the two MAC outputs will collide.

Unfortunately, this method does not work because the lengths of the messages are different, and thus the last message block with the length padding will be different and prevent the cycle collision to propagate to the MAC output. We will use a trick to overcome this issue, even though the basic method remains the same. The idea is to build a message M going twice inside the cycle of the giant component, so that we can add L [0] message blocks in the first cycle to obtain a message M_1 and L [0] message blocks in the second cycle to obtain M_2. This is depicted in Figure 2: M_1 will cycle L [0] message blocks in the first cycle (red dotted arrows), while M_2 will cycle L [0] message blocks in the second cycle (blue dashed arrows), and thus they will both have the same length overall.

To perform the distinguishing-H attack, the adversary simply randomly selects an initial message block m and query $M_1 = m \parallel [0]^{2^{l/2}} \parallel [1] \parallel [0]^{2^{l/2}+L}$ and $M_2 = m \parallel [0]^{2^{l/2}+L} \parallel [1] \parallel [0]^{2^{l/2}}$ to the MAC oracle, and deduce that the target function is used to instantiate the oracle if the two MAC values are colliding. The [1] message block is used to quit the cycle and randomize the entry point to return again in the giant component. We give below a detailed attack procedure and complexity analysis.

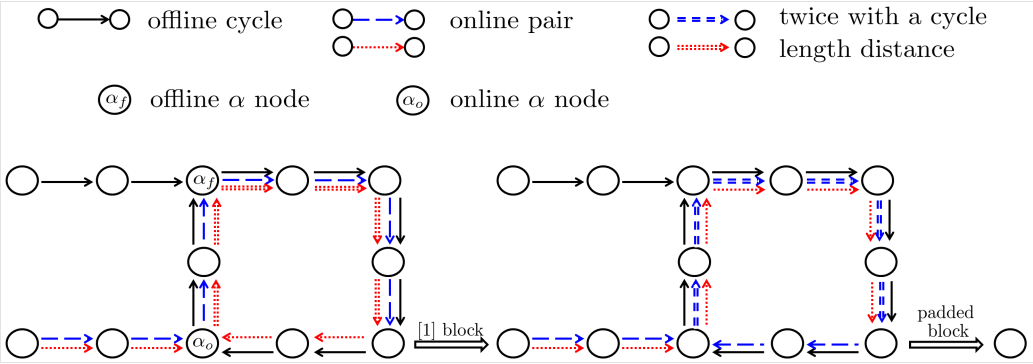


Fig. 2. Distinguishing-H attack

This attack is very interesting as the first generic distinguishing-H attack on `HMAC` and `NMAC` with a complexity lower than 2^l. However, we note that the very long message length might be a limitation. In theory this is of no importance and our attack is indeed valid, but in practice some hash functions forbid message inputs longer than a certain length. To address this issue we provide an alternative attack in the full version of this paper, using shorter messages, at the cost of a higher complexity.

4.2 Detailed attack process

1. (offline) Search for a cycle in the functional graph of $h_{[0]}$ and denote L its length.
2. (online) Choose a random message block m and query the `HMAC` value of the two messages $M_1 = m\|[0]^{2^{l/2}}\|[1]\|[0]^{2^{l/2}+L}$ and $M_2 = m\|[0]^{2^{l/2}+L}\|[1]\|[0]^{2^{l/2}}$.
3. If the `HMAC` values of M_1 and M_2 collide then output 1, otherwise output 0.

4.3 Complexity and success probability analysis

We would like to evaluate the complexity of the attack. The first step will require about $2^{l/2}$ offline computations to find the cycle for the target compression function h. It is important to note that we can run this first step several times

in order to ensure that we are using the cycle from the largest component of the functional graph of $h_{[0]}$. The second step makes two queries of about $2^{l/2} + 2^{l/2} + L \simeq 3 \cdot 2^{l/2}$ message blocks each. Therefore, the overall complexity of the attack is about $2^{l/2+3}$ compression function computations.

Next, we evaluate the advantage of the adversary in winning the distinguishing-H game defined in Section 2.2. We start with the case where the oracle is instantiated with the real compression function h. The adversary will output 1 if a collision happens between the HMAC computations of M_1 and M_2. Such a collision can happen when the following conditions are satisfied:

- The processing of the random block m sets us in the same component (of the functional graph of $h_{[0]}$) as in the offline computation. Since we ensured that the largest component was found during the offline computation, and since it has an average size of $0.7582 \cdot 2^l$ elements according to Theorem 2, this event will happen with probability 0.7582.
- The $2^{l/2}$ [0] message blocks concatenated after m are enough to reach the cycle of the component, i.e. the tail length when reaching the first cycle is smaller than $2^{l/2}$. Since the average tail length is less than $2^{l/2}$ elements, this will happen with probability more than 1/2.
- The processing of the block [1] sets us in the same component (of the functional graph of $h_{[0]}$) as in the offline computation; again the probability is 0.7582.
- The $2^{l/2}$ [0] message blocks concatenated after [1] are enough to reach the cycle of the component; again this happens with probability 1/2.

The collision probability is then $(0.7582)^2 \times 1/4 \simeq 0.14$ and thus we have that $\Pr[\mathcal{A}(\mathtt{HMAC}_K^h) = 1] \geq 0.14$. In the case where the oracle is instantiated with a random function r, the adversary will output 1 if and only if a random HMAC collision happens between messages M_1 and M_2. Such a collision can be obtained if the processing of any of the last $[0]^{2^{l/2}}$ blocks of the two messages leads to an internal collision, therefore with negligible probability $2^{l/2} \cdot 2^{-l} = 2^{-l/2}$. Overall, the adversary advantage is equal to $Adv(\mathcal{A}) = \left|0.14 - 2^{-l/2}\right| \simeq 0.14$.

5 Internal state recovery for NMAC and HMAC

This section extends the distinguishing-H attack in order to build an internal state recovery attack.

5.1 General description

In order to extent the distinguishing-H attack to a state recovery attack, we observe that there is a high probability that the α-node reached in the online phase is the root of the giant tree of the functional graph of $h_{[0]}$. More precisely, we can locate the largest tree and the corresponding α-node in the offline phase, by repeating the cycle search a few times. We note that $\delta_1 + \delta_2 > 1$, therefore the largest tree is in the largest component with asymptotic probability one (see

Theorem 3). Thus, assuming that the online phase succeeds, the α-node reached in the online phase is the root of the largest tree with asymptotic probability $\delta_1/\delta_2 \approx 0.63$. If we can locate the α-node of the online phase (i.e. we deduce its block index inside the queried message), we immediately get its corresponding internal state value from the offline computations.

Since the average rho length is $\sqrt{\pi 2^l/2}$ we can not use a brute-force search to locate the α-node, but we can use a binary search instead. We denote the length of the cycle of the giant component as L, and the follow the distinguishing-H attack to reach the cycle. We choose a random message block m and we query the two messages $M_1 = m \,\|\, [0]^{2^{l/2}} \,\|\, [1] \,\|\, [0]^{2^{l/2}+L}$ and $M_2 = m \,\|\, [0]^{2^{l/2}+L} \,\|\, [1] \,\|\, [0]^{2^{l/2}}$. If the MAC collide, we know that the state reached after processing $m \,\|\, [0]^{2^{l/2}}$ is located inside the main cycle. We use a binary search to find the smallest X so that the state reached after $m \,\|\, [0]^X$ is in the cycle.

The first step of the binary search should decide whether the node reached after $m \,\|\, [0]^{2^{l/2-1}}$ is also inside the first online cycle or not. More precisely, we check if the two messages $M_1' = m \,\|\, [0]^{2^{l/2-1}} \,\|\, [1] \,\|\, [0]^{2^{l/2}+L}$ and $M_2' = m \,\|\, [0]^{2^{l/2-1}+L} \,\|\, [1] \,\|\, [0]^{2^{l/2}}$ also give a colliding tag. If it is the case, then the node after processing $m \,\|\, 2^{l/2-1}$ is surely inside the cycle, which implies that the α-node is necessarily located in the first $2^{l/2-1}$ zero blocks. On the other hand, if the tags are not colliding, we cannot directly conclude that the α-node is located in the second half since there is a non-negligible probability that the α-node is in the first half but the [1] block is directing the paths in M_1' and M_2' to distinct components in the functional graph of $h_{[0]}$ (in which case the tag are very likely to differ). Therefore, we have to test for collisions with several couples $M_1^u = m \,\|\, [0]^{2^{l/2-1}} \,\|\, [u] \,\|\, [0]^{2^{l/2}+L}$, $M_2^u = m \,\|\, [0]^{2^{l/2-1}+L} \,\|\, [u] \,\|\, [0]^{2^{l/2}}$, and if none of them collide we can safely deduce that the α-node is located in the second $2^{l/2-1}$ zero blocks. Overall, one such step reduces the number of the candidate nodes by a half and we simply continue this binary search in order to eventually obtain the position of the α-node with $\log_2(2^{l/2}) = l/2$ iterations.

5.2 Detailed attack process

1. (offline) Search for a cycle in the functional graph of $h_{[0]}$ and denote L its length.
2. (online) Find a message block m such that querying the two messages $M_1 = m \,\|\, [0]^{2^{l/2}} \,\|\, [1] \,\|\, [0]^{2^{l/2}+L}$ and $M_2 = m \,\|\, [0]^{2^{l/2}+L} \,\|\, [1] \,\|\, [0]^{2^{l/2}}$ leads to the same `HMAC` output. Let X_1 and X_2 be two integer variables, initialized to the values 0 and $2^{l/2}$ respectively.
3. (online) Let $X' = (X_1 + X_2)/2$. Select $\beta \log(l)$ distinct message blocks $[u]$, and for each of them query the `HMAC` output for messages $M_1^u = m \,\|\, [0]^{X'} \,\|\, [u] \,\|\, [0]^{2^{l/2}+L}$ and $M_2^u = m \,\|\, [0]^{X'+L} \,\|\, [u] \,\|\, [0]^{2^{l/2}}$. If at least one of the (M_1^u, M_2^u) pairs leads to a colliding `HMAC` output, then set $X_2 = X'$. Otherwise, set $X_1 = X'$. We use $\beta = 4.5$ as explained later.
4. (online) If $X_1 + 1 = X_2$ holds, output X_2 as the block index of the α-node. Otherwise, go back to the previous step.

5.3 Complexity and success probability analysis

Complexity. We would like to evaluate the complexity of the attack. The first step will require about $2^{l/2}$ offline computations to find the cycle. Again, it is important to note that we can run this first step several times in order to ensure that we are using the cycle from the biggest component of the functional graph of $h_{[0]}$. The second step repeats the execution of the distinguishing-H attack from Section 4, which requires $6 \cdot 2^{l/2}$ computations for a success probability of 0.14, until it finds a succeeding message block m. Therefore, after trying a few m values, we have probability very close to 1 to find a valid one. The third and fourth steps will be executed about $l/2$ times (for the binary search), and each iteration of the third step performs $2 \cdot \beta \log(l)$ queries of about $2^{l/2} + 2^{l/2} + L \simeq 3 \cdot 2^{l/2}$ message blocks each. Therefore, the overall complexity of the attack is about $3\beta \cdot l \cdot \log(l) \cdot 2^{l/2}$ compression function computations.

Success probability. Next we evaluate the success probability that the attacker recovers the internal state and this depends on the success probability of the binary search steps. We start with the case where the node after $m \parallel [0]^{X'}$ is inside the first online cycle. The third step will succeed as long as at least one of the (M_1^u, M_2^u) pairs collide on the output (we can omit the false positive collisions which happen with negligible probability). One pair (M_1^u, M_2^u) will indeed collide if:

- The random block $[u]$ sends both messages to the main component of the functional graph of $h_{[0]}$. Since it has an expected size of $\delta_2 \cdot 2^l$ (see Theorem 3), this is the case with probability δ_2^2.
- The $2^{l/2}$ $[0]$ message blocks concatenated after $[u]$ are enough to reach the cycle, i.e. the tail length when reaching the second cycle is smaller than $2^{l/2}$. Since the average tail length is smaller than $2^{l/2}$ elements, this will happen with probability $1/2$ for each message.

After trying $\beta \log(l)$ pairs, the probability that at least one pair collides is $1 - (1 - \delta_2^2/4)^{\beta \log(l)}$. If we use $\beta = -1/\log(1 - \delta_2^2/4) \approx 4.5$, this gives a probability of $1 - 1/l$. On the other hand, if the node after $m \parallel [0]^{X'}$ is not inside the cycle, the third step will succeed when no random collision occurs among the $\beta \log(l)$ tries, and such collisions happen with negligible probability. Overall, since there are $l/2$ steps in the binary search, the average success probability of the binary search is $(1 - 1/l)^{l/2} \geq e^{-1/2} \approx 0.6$.

Finally, the attack succeeds if the α-node is actually the root of the giant tree, as computed in the offline phase. This is the case with probability δ_1/δ_2, and the success probability of the full state recovery attack is $\delta_1/\delta_2 \cdot e^{-1/2} \approx 0.38$.

6 Internal state recovery with shorter messages

In the full version of the paper, we give a an alternative internal-state recovery attack using shorter messages, that can also be used as a distinguishing-H attack

with shorter messages. The attacks of Sections 4 and 5 have a complexity of $O(2^{l/2})$ using a small number of messages of length $2^{l/2}$; on the other hand the alternative attack has a complexity $O(2^{3l/4})$ using $2^{l/2}$ messages of length $2^{l/4}$. More generally, if the message size is limited to 2^s blocks ($s \leq l/4$), then the attack requires 2^{l-2s} messages. Due to space constraints, we only give a brief description of this attack here.

6.1 Entropy loss in collisions finding

While the previous attacks are based on detecting cycles in the graph of a random function, this alternative attack is based on the fact that finding collisions by iterating a random function does not give a random collision: some particular collisions are much more likely to be found than others. This fact is well known in the context of collision search; for instance van Oorschot and Wiener [23] recommend to randomize the function regularly when looking for a large number of collisions. In this attack, we exploit this property to our advantage: first we use a collision finding algorithm to locate collisions in h_M with a fixed M; then we query the MAC oracle with messages with long repetitions of the block M and we detect collisions in the internal state; since the collisions found in this way are not randomly distributed, there is a good probability that we will reach one the collisions that was previously detected in the offline phase.

Actually, the attacks of Sections 4 and 5 can also be seen as following this stategy: we use a collision finding algorithm based on cycle detection (following Pollard's rho algorithm), and we know that with a good probability, the collision found will be the root of the giant tree. For the alternative attack, we use a collision finding algorithm similar to [23], but using using fixed length chains. In the full version of the paper, we study the entropy of the distribution of collisions found in this way, and we show that when using chains of length 2^s, we need about $2^{l/2-s}$ collisions in order to have a match between the online and offline steps. This translates to an attack complexity of 2^{l-s}, with $s \leq l/4$.

7 Key recovery for HMAC based on a hash function with an internal checksum

In this section we study HMAC used with a hash function with an internal checksum, such as GOST. We first show that the checksum does not prevent the distinguishing-H and state recovery attack, but more surprisingly the checksum actually allows to mount a full key-recovery attack significantly more efficient than exhaustive search.

A hash function with an internal checksum computes the sum of all message blocks, and uses this sum as an extra input to the finalization function. The sum can be computed for any group operation, but it will usually be an XOR sum or a modular addition. We use the XOR sum $\texttt{Sum}^{\oplus}$ to present our attack, but it is applicable with any group operation.

The checksum technique has been used to enhance the security of a hash function, assuming that controlling the checksum would be an additional challenge for an adversary. While previous work argued its effectiveness [10], our result reveals a surprising fact that incorporating a checksum into a hash function could even weaken its security in some applications such as HMAC.

A notable example of a hash function with a checksum is the GOST hash function, which has been standardized by the Russian Government [6] and by IETF [5]. HMAC-GOST has also been standardized by IETF [15] and is implemented in OpenSSL. GOST uses parameters $n = l = b$, and uses a separate call to process the message length, as follows:

$$\begin{array}{lll} x_0 = IV & x_{i+1} = h(x_i, m_i) & x_* = h(x_p, |M|) \\ \sigma_0 = 0 & \sigma_{i+1} = \sigma_i \oplus m_i & \mathsf{hash} = g(x_*, \sigma_p) \end{array}$$

If this section we describe the attack on GOST-like functions following this structure; Figure 3 shows an HMAC computation with a GOST-like hash function. We give more general attacks when the output is computed as $g(x_p, |M|, \sigma_p)$ in the full version of this paper.

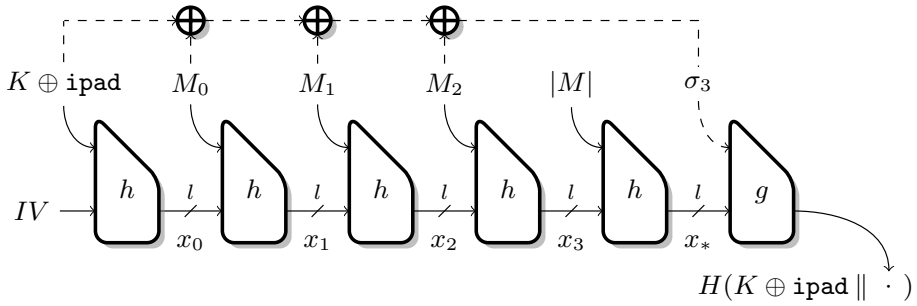


Fig. 3. HMAC based on a hash function with a checksum (dashed lines) and a length-padding block. We only detail the first hash function call.

7.1 General description

In HMAC, $K \oplus \texttt{ipad}$ is prepended to a message M, and $(K \oplus \texttt{ipad}) \,\|\, M$ is hashed by the underlying hash function H. Therefore, the final checksum value is $\sigma_p = \texttt{Sum}^{\oplus}((K \oplus \texttt{ipad}) \,\|\, M) = K \oplus \texttt{ipad} \oplus \texttt{Sum}^{\oplus}(M)$. In this attack, we use the state recovery attack to recover the internal state x_* before the checksum is used and we take advantage of the fact that the value σ_p actually contains the key, but can still be controlled by changing the message. We use this to inject a known difference in the checksum, and to perform a kind of related key attack on the finalization function g, even though we have access to a single HMAC key.

More precisely, we use Joux's multicollision attack [11] to generate a large set of messages with the same value $\bar{x}$ for x_*, but with different values of the

checksum. We detect MAC-collisions among those messages, and we assume that the collisions happens when processing the checksum of the internal hash function. For each such collision, we have $g(\bar{x}, K \oplus \texttt{ipad} \oplus \texttt{Sum}^{\oplus}(M)) = g(\bar{x}, K \oplus \texttt{ipad} \oplus \texttt{Sum}^{\oplus}(M'))$, and we compute the input difference $\Delta M = \texttt{Sum}^{\oplus}(M) \oplus \texttt{Sum}^{\oplus}(M')$.

Finally, we compute $g(\bar{x}, m)$ offline for a large set of random values m, and we collect collisions. Again, we compute the input difference $\Delta m = m \oplus m'$ for each collision, and we match Δm to the previously stored ΔM. When a match is found between the differences we look for the corresponding values and we have $K \oplus \texttt{ipad} \oplus \texttt{Sum}^{\oplus}(M) = m$ (or m') with high probability. This gives the value of the key K.

State recovery with a checksum. First, we note that the checksum σ does not prevent the state recovery attacks of Section 5; the complexity only depend on the size l of the state x. Indeed, the attack of Section 5 is based on detecting collisions between pairs of messages $M_1 = m \,\|\, [0]^{2^{l/2}} \,\|\, [k] \,\|\, [0]^{2^{l/2}+L}$ and $M_2 = m \,\|\, [0]^{2^{l/2}+L} \,\|\, [k] \,\|\, [0]^{2^{l/2}}$. Since the messages have the same checksum, a collision in the state will be preserved. More generally, the attacks can easily be adapted to use only message with a fixed sum. For instance, we can use random messages with two identical blocks in the attack of Section 5, and messages of the form $m \,\|\, m \,\|\, [0]^{2^{l/2}} \,\|\, [k] \,\|\, [k] \,\|\, [0]^{2^{l/2}+L}$ have a checksum of zero.

Recovering the state of a short message. Unfortunately, the state we recover will correspond to a rather long message (*e.g.* $2^{l/2}$ blocks), and all the queries based on this message will be expensive. In order to overcome this issue, we use the known state x_M after a long message M to recover the state after a short one. More precisely, we generate a set of $2^{l/4}$ long messages by appending a random message blocks m twice to M. Note that $\texttt{Sum}^{\oplus}(M \,\|\, m \,\|\, m) = \texttt{Sum}^{\oplus}(M)$. Meanwhile, we generate a set of $2^{3l/4}$ two-block messages $m_1 \,\|\, m_2$, with $m_1 \oplus m_2 = \texttt{Sum}^{\oplus}(M)$. We query these two sets to the `HMAC` oracle and collect collisions between a long and a short message. We expect that one collision correspond to a collision in the value x_* before the finalization function g. We can compute the value x_* for the long message from the known state x_M after processing M. This will correspond to the state after processing the message $m_1 \,\|\, m_2$ and its padding block, or equivalently, after processing the message $m_1 \,\|\, m_2 \,\|\, [2]$ (because the length block is processed with the same compression function). We can verify that the state is correctly recovered by generating a collision $m \,\|\, m, m' \,\|\, m'$ offline from the state x_*, and comparing $\texttt{HMAC}(m_1 \,\|\, m_2 \,\|\, [2] \,\|\, m \,\|\, m)$ and $\texttt{HMAC}(m_1 \,\|\, m_2 \,\|\, [2] \,\|\, m' \,\|\, m')$.

7.2 Detailed attack process

For simplicity of the description, we omit the padding block in the following description, and we refer to the previous paragraphs for the details of how to deal with the padding.

1. Recover an internal state value x_r after processing a message M_r through HMAC. Refer to Section 5 for the detailed process.
2. (online) Choose $2^{l/4}$ one-block random messages m, query $M_r \parallel m \parallel m$ to HMAC and store m and the corresponding tag.
3. (online) Choose $2^{3l/4}$ one-block random messages m, query $(\mathtt{Sum}^{\oplus}(M_r) \oplus m) \parallel m$ to HMAC and look for a match between the tag value and one of the stored tag values in Step 2.
   For a colliding pair $(\mathtt{Sum}^{\oplus}(M_r) \oplus m) \| m$ and $M_r \| m' \| m'$, denote $\mathtt{Sum}^{\oplus}(M_r) \oplus m \| m$ as M_1 and $h(h(x_r, m'), m')$ as x_1. Generate a collision $h(h(x_1, u), u) = h(h(x_1, u'), u')$. Query $M_1 \parallel u \parallel u$ and $M_1 \parallel u' \parallel u'$ and compare the tags. If they are equal, the internal state after processing M_1 (before the checksum block) is x_1.
4. (offline) Generate $2^{3l/4}$ messages that all collide on the internal state before the checksum block by Joux's multicollision. More precisely, choose $2^{l/2}$ random message m and compute $h(x_1, m)$ to find a collision $h(x_1, m_1) = h(x_1, m'_1) = x_2$. Then iterate this procedure to find a collision $h(x_i, m_i) = h(x_i, m'_i) = x_{i+1}$ for $i \leq 3l/4$. Denote the value of $x_{3l/4+1}$ by $\bar{x}$.
5. (online) Query the set of messages in Step 4 to HMAC in order to collect tag collisions. For each collision M and M', compute the checksum difference $\Delta M = \mathtt{Sum}^{\oplus}(M) \oplus \mathtt{Sum}^{\oplus}(M')$, and store $(\mathtt{Sum}^{\oplus}(M), \Delta M)$.
6. (offline) Choose a set of $2^{3l/4}$ one-block random message m, compute $g(\bar{x}, m)$ and collect collisions. For each collision m and m', compute the difference $\Delta m = m \oplus m'$ and match Δm to the stored ΔM at Step 5. If a match is found, mark $\mathtt{Sum}^{\oplus}(M) \oplus \mathtt{ipad} \oplus m$ and $\mathtt{Sum}^{\oplus}(M) \oplus \mathtt{ipad} \oplus m \oplus \Delta m$ as potential key candidates.
7. (offline) filter the correct key from the potential candidates by verifying a valid message/tag pair.

7.3 Complexity and success probability analysis

We need to evaluate the complexity of our key recovery attack.

Step 1:	$O(l \cdot \log(l) \cdot 2^{l/2})$	**Step 2:**	$2^{3l/4}$	**Step 3:**	$2 \cdot 2^{3l/4}$
Step 4:	$3l/4 \cdot 2^{l/2}$	**Step 5:**	$3l/4 \cdot 2^{3l/4}$	**Step 6:**	$2^{3l/4}$
Step 7:	$O(1)$				

Overall, the fifth step dominates the complexity, and the total complexity is about $3l/4 \cdot 2^{3l/4}$ compression function computations.

Next we evaluate the success probability of our method. The first step succeeds with a probability almost 1 after several trials. Steps 2 and 3 need to guarantee a collision between a long and a short message. Since there are 2^l pairs, one such collision occurs with a probability of $1 - (1 - 2^{-l})^{2^l} \approx 1 - 1/e \approx 0.63$. The success probability of producing no less than $2^{l/2}$ collisions at each of steps 5 and 6 is 0.5 since the expected number of collisions is $2^{l/2}$. Thus the overall success probability is no less than 0.16.

Conclusion

Our results show that the security of HMAC and hash-based MAC above the birthday bound is significantly weaker than previously expected. First, we show that distinguishing-H and state-recovery attacks are not much harder than a distinguishing-R attack, contrary to previous beliefs. Second, we show that the use of a checksum can allow a key-recovery attack against HMAC with complexity only $\tilde{O}(2^{3l/4})$. In particular, this attack is applicable to HMAC-GOST, a standardized construction.

We give a comparison of our attacks and previous attack against concrete instances of HMAC in Table 1, showing that some attacks against concrete instances are in fact less efficient than our generic attacks.

As future works, it would be interesting to find other applications of the internal state recovery for HMAC. Moreover, we expect further applications of the analysis of the functional graph, as it might be possible to use other distinguishing properties, such as the tail length, the distance of a node from the cycle, etc.

Acknowledgments

The authors would like to thank the anonymous referees for their helpful comments. Gaëtan Leurent is supported by the ERC project CRASH. Thomas Peyrin and Lei Wang are supported by the Singapore National Research Foundation Fellowship 2012 (NRF-NRFF2012-06).

References

1. Bellare, M.: New Proofs for NMAC and HMAC: Security without Collision-Resistance. In Dwork, C., ed.: CRYPTO. Volume 4117 of Lecture Notes in Computer Science., Springer (2006) 602–619
2. Bellare, M., Canetti, R., Krawczyk, H.: Keying Hash Functions for Message Authentication. In Koblitz, N., ed.: CRYPTO. Volume 1109 of Lecture Notes in Computer Science., Springer (1996) 1–15
3. Brassard, G., ed.: Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings. In Brassard, G., ed.: CRYPTO. Volume 435 of Lecture Notes in Computer Science., Springer (1990)
4. Damgård, I.: A Design Principle for Hash Functions. [3] 416–427
5. Dolmatov, V.: GOST R 34.11-94: Hash Function Algorithm. RFC 5831 (Informational) (March 2010)
6. FAPSI, VNIIstandart: GOST 34.11-94, Information Technology Cryptographic Data Security Hashing Function (in Russian) (1994)
7. Flajolet, P., Odlyzko, A.M.: Random Mapping Statistics. In Quisquater, J.J., Vandewalle, J., eds.: EUROCRYPT. Volume 434 of Lecture Notes in Computer Science., Springer (1989) 329–354
8. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press (2009)

9. Fouque, P.A., Leurent, G., Nguyen, P.Q.: Full Key-Recovery Attacks on HMAC/NMAC-MD4 and NMAC-MD5. In Menezes, A., ed.: CRYPTO. Volume 4622 of Lecture Notes in Computer Science., Springer (2007) 13–30
10. Gauravaram, P., Kelsey, J.: Linear-XOR and Additive Checksums Don't Protect Damgård-Merkle Hashes from Generic Attacks. In Malkin, T., ed.: CT-RSA. Volume 4964 of Lecture Notes in Computer Science., Springer (2008) 36–51
11. Joux, A.: Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In Franklin, M.K., ed.: CRYPTO. Volume 3152 of Lecture Notes in Computer Science., Springer (2004) 306–316
12. Kim, J., Biryukov, A., Preneel, B., Hong, S.: On the Security of HMAC and NMAC Based on HAVAL, MD4, MD5, SHA-0 and SHA-1 (Extended Abstract). In Prisco, R.D., Yung, M., eds.: SCN. Volume 4116 of Lecture Notes in Computer Science., Springer (2006) 242–256
13. Merkle, R.C.: One Way Hash Functions and DES. [3] 428–446
14. Peyrin, T., Sasaki, Y., Wang, L.: Generic Related-Key Attacks for HMAC. In Wang, X., Sako, K., eds.: ASIACRYPT. Volume 7658 of Lecture Notes in Computer Science., Springer (2012) 580–597
15. Popov, V., Kurepkin, I., Leontiev, S.: Additional Cryptographic Algorithms for Use with GOST 28147-89, GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms. RFC 4357 (Informational) (January 2006)
16. Preneel, B.: HMAC. In van Tilborg, H.C.A., Jajodia, S., eds.: Encyclopedia of Cryptography and Security (2nd Ed.). Springer (2011) 559–560
17. Preneel, B., van Oorschot, P.C.: MDx-MAC and Building Fast MACs from Hash Functions. In Coppersmith, D., ed.: CRYPTO. Volume 963 of Lecture Notes in Computer Science., Springer (1995) 1–14
18. Rechberger, C., Rijmen, V.: New Results on NMAC/HMAC when Instantiated with Popular Hash Functions. J. UCS **14**(3) (2008) 347–376
19. Rivest, R.L.: The MD4 Message Digest Algorithm. In Menezes, A., Vanstone, S.A., eds.: CRYPTO. Volume 537 of Lecture Notes in Computer Science., Springer (1990) 303–311
20. Rivest, R.L.: The MD5 message-digest algorithm. Request for Comments (RFC) 1320, Internet Activities Board, Internet Privacy Task Force (April 1992)
21. Tsudik, G.: Message Authentication with One-Way Hash Functions. In: INFOCOM. (1992) 2055–2059
22. U.S. Department of Commerce, National Institute of Standards and Technology: Secure Hash Standard (SHS) (Federal Information Processing Standards Publication 180-3). (2008) `http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf`.
23. van Oorschot, P.C., Wiener, M.J.: Parallel Collision Search with Cryptanalytic Applications. J. Cryptology **12**(1) (1999) 1–28
24. Wang, L., Ohta, K., Kunihiro, N.: New Key-Recovery Attacks on HMAC/NMAC-MD4 and NMAC-MD5. In Smart, N.P., ed.: EUROCRYPT. Volume 4965 of Lecture Notes in Computer Science., Springer (2008) 237–253
25. Wang, X., Yu, H., Wang, W., Zhang, H., Zhan, T.: Cryptanalysis on HMAC/NMAC-MD5 and MD5-MAC. In Joux, A., ed.: EUROCRYPT. Volume 5479 of Lecture Notes in Computer Science., Springer (2009) 121–133
26. Yasuda, K.: "Sandwich" Is Indeed Secure: How to Authenticate a Message with Just One Hashing. In Pieprzyk, J., Ghodosi, H., Dawson, E., eds.: ACISP. Volume 4586 of Lecture Notes in Computer Science., Springer (2007) 355–369

27. Yu, H., Wang, X.: Full Key-Recovery Attack on the HMAC/NMAC Based on 3 and 4-Pass HAVAL. In Bao, F., Li, H., Wang, G., eds.: ISPEC. Volume 5451 of Lecture Notes in Computer Science., Springer (2009) 285–297

A Additional Figures

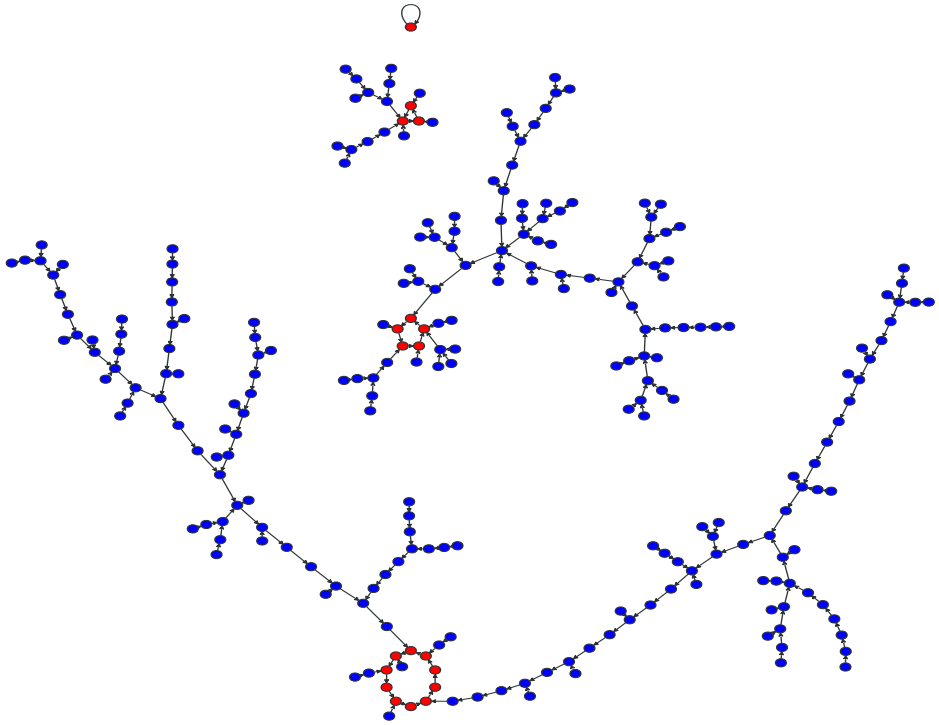


Fig. 4. Functional graph of Keccak (`SHA-3`) with 8-bit input and 8-bit output.

Improved Generic Attacks Against Hash-based MACs and HAIFA

Itai Dinur · Gaëtan Leurent

Received: 1 July 2015 / Accepted: 21 October 2016
This is the author version submitted by the authors .
The published version is available at DOI 10.1007/s00453-016-0236-6.

Abstract The security of `HMAC` (and more general hash-based MACs) against state-recovery and universal forgery attacks was shown to be suboptimal, following a series of results by Leurent *et al.* and Peyrin *et al.*. These results have shown that such powerful attacks require significantly less than 2^ℓ computations, contradicting the common belief (where ℓ denotes the internal state size). In this work, we revisit and extend these results, with a focus on concrete hash functions that limit the message length, and apply special iteration modes.

We begin by devising the first state-recovery attack on `HMAC` with a HAIFA hash function (using a block counter in every compression function call), with complexity $2^{4\ell/5}$. Then, we describe improved tradeoffs between the message length and the complexity of a state-recovery attack on `HMAC` with a Merkle-Damgård hash function. Consequently, we obtain improved attacks on several `HMAC` constructions used in practice, in which the hash functions limits the maximal message length (e.g., `SHA-1` and `SHA-2`). Finally, we present the first universal forgery attacks, which can be applied with short message queries to the MAC oracle. In particular, we devise the first universal forgery attacks applicable to `SHA-1` and `SHA-2`.

Despite their theoretical interest, our attacks do not seem to threaten the practical security of the analyzed concrete `HMAC` constructions.

Keywords Hash functions · MAC · `HMAC` · Merkle-Damgård · HAIFA · state-recovery attack · universal forgery attack · `GOST` · `Streebog` · `SHA` family

This paper in an extended version of [6], presented at CRYPTO 2014.

Itai Dinur
Department of Computer Science, Ben-Gurion University, Beer-Sheva, Israel
E-mail: dinuri@cs.bgu.ac.il

Gaëtan Leurent
Inria, EPI SECRET, France
E-mail: Gaetan.Leurent@inria.fr

1 Introduction

MAC algorithms are an important symmetric cryptography primitive, used to verify the integrity and authenticity of messages. The sender of a message uses a MAC function to compute a tag from the message and a shared secret key. The tag is appended to the message and the receiver can recompute the tag using the key, and reject the message when it does not match the received one. The main security requirement of a MAC is the resistance to existential forgery. Namely, after querying the MAC oracle to obtain the tags of some carefully chosen messages, it should be hard for an adversary to forge a valid tag for a different message.

One of the most widely used MAC algorithms in practice is `HMAC`, a MAC construction using a hash function designed by Bellare, Canetti and Krawczyk in 1996 [4]. The algorithm has been standardized by ANSI, IETF, ISO and NIST, and is widely deployed to secure internet communications (*e.g.* SSL/TLS, SSH, IPSec). As these protocols are widely used, the security of `HMAC` has been extensively studied, and several security proofs [3, 4] show that it gives a secure MAC and a secure PRF up to the birthday bound (assuming good properties of the underlying compression function). At the same time, there is a simple existential forgery attack on any iterative MAC with an ℓ-bit state, with complexity $2^{\ell/2}$, matching the security proof. Nevertheless, security beyond the birthday bound for stronger attacks (such as state-recovery and universal forgery) is still an important topic.

Surprisingly, the security of `HMAC` beyond the birthday bound has not been thoroughly studied until 2012, when Peyrin and Sasaki described an attack on `HMAC` in the related-key setting [23]. Later work focused on single-key security, and included a paper by Naito, Sasaki, Wang and Yasuda [21], which described state-recovery attacks with complexity $2^{\ell}/\ell$. At Asiacrypt 2013, Leurent, Peyrin and Wang [19] gave state-recovery attacks with complexity $2^{\ell/2}$, closing the gap with the security proof. Later, at Eurocrypt 2014, Peyrin and Wang [24] described a universal forgery attack with complexity as low as $2^{5\ell/6}$, a result that was further improved to $2^{3\ell/4}$ at CRYPTO 2014 [11], showing that even this very strong attack is possible with significantly less than 2^{ℓ} work.

Some of the generic attacks have also been used as a first step to build specific attacks against `HMAC` with the concrete hash function Whirlpool [12, 13].

These results show that more work is needed to better understand the exact security provided by `HMAC` and hash-based MACs in general.

1.1 Our results

In this paper, we provide several important contributions to the security analysis of `HMAC` and similar hash-based MAC constructions. In particular, we devise improved attacks when `HMAC` is used with many popular concrete hash functions, and in several cases our attacks are the first to be applicable to `HMAC`

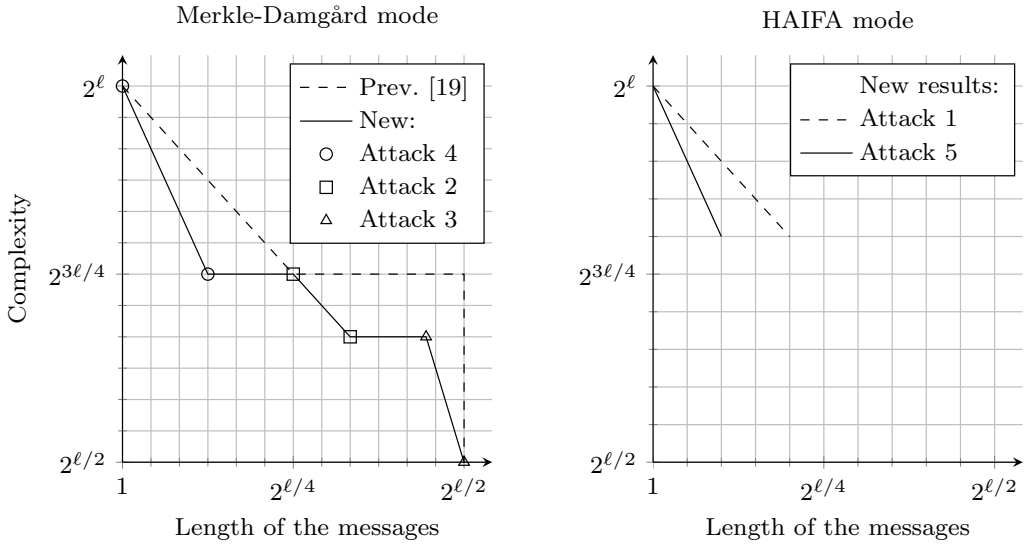


Fig. 1 Tradeoffs between the message length and the complexity for state-recovery attacks

with the given hash function. Some results with concrete instantiations are summarized in Table 1.[1]

As a first contribution, we focus on the HAIFA [5] mode of operation, used in many hash function designs such as BLAKE [1,2], Skein [8], or Streebog [7]. The HAIFA construction uses a block counter to tweak the compression functions, such that they resemble independent random functions, in order to thwart some narrow-pipe attacks (*e.g.* the second-preimage attack of Kelsey and Schneier [16]). Indeed, the previous attacks against `HMAC` [19,24] use in a very strong way the assumption that the same compression function is applied to all the message blocks, and thus they cannot be applied to HAIFA. In this work, we present the first state-recovery attack on `HMAC` using these hash functions, whose optimal complexity is $2^{4\ell/5}$.

In an interesting application of our state-recovery attack on HAIFA (given in Section 8), we show how to extend it into a key-recovery attack on the new Russian standard `Streebog`, recovering the 512-bit key of `HMAC-Streebog` with a complexity of 2^{417}. This key-recovery attack is similar to the one of [19] for Merkle-Damgård, and confirms its surprising observation: adding internal checksums in a hash function (such as `Streebog`) *weakens* the design when used in `HMAC`, even for hash functions based on the HAIFA mode.

As a second contribution of this paper, we revisit the results of the full version of [19] for Merkle-Damgård hash functions (given in [18]), and we prove the conjectures used in its short message attacks. Some of our proofs are of independent interest, as they give insight into the behavior of classical collision search algorithms for random functions. These proofs explain for the first time an interesting phenomenon experimentally observed in several previous works

[1] We elaborate on our complexity evaluation in the next Subsection.

Table 1 Complexity of attacks on HMAC instantiated with some concrete hash functions. The state size is denoted as ℓ, and the maximum message length as 2^s. For the new results, we give a reference to the Attack number. The figures omit small constants (following the analysis in this paper and previous works), but take into account polynomial factors in ℓ.

				State-recovery		Universal forgery	
Function	Mode	ℓ	s	[19]	New	[24]	New
SHA-1	MD	160	2^{55}	2^{120}	2^{107} (2)	N/A	2^{132} (8)
SHA-224	MD	256	2^{55}	2^{201}	2^{192} (4)	N/A	N/A
SHA-256	MD	256	2^{55}	2^{201}	2^{192} (4)	N/A	2^{228} (7,8)
SHA-512	MD	512	2^{118}	2^{394}	2^{384} (4)	N/A	2^{453} (7,8)
HAVAL	MD	256	2^{54}	2^{202}	2^{192} (4)	N/A	2^{229} (7,8)
WHIRLPOOL	MD	512	2^{247}	2^{384}	2^{283} (3)	N/A	2^{419} (8)
BLAKE-256	HAIFA	256	2^{55}	N/A	2^{213} (5)	N/A	N/A
BLAKE-512	HAIFA	512	2^{118}	N/A	2^{419} (5)	N/A	N/A
Skein-512	HAIFA	512	2^{90}	N/A	2^{419} (5)	N/A	N/A
						Key-recovery	
						[19]	New
Streebog	HAIFA+σ	512	∞	N/A	2^{417} (5,B)	N/A	2^{417} (6,B)

(such as [22]), namely, that the collisions found by such algorithms are likely to belong to a restricted set of a surprisingly small size.

Then, based on our proofs, we describe several new algorithms with various improved tradeoffs between the message length and the complexity as shown in Figure 1. As many concrete hash functions restrict the message size, we obtain improved attacks in many cases: for instance, we reduce the complexity of a state-recovery attack against HMAC-SHA-1 from 2^{120} to 2^{107} and HMAC-WHIRLPOOL from 2^{384} to 2^{283} (see Table 1).

Finally, we focus on universal forgery attacks, where the previous attacks of [11,24] are much more efficient than exhaustive search, but require in an inherent way to query the MAC oracle with very long messages of about $2^{\ell/2}$ blocks. Thus, these attacks cannot be applied to many concrete hash functions that limit the message size. On the other hand, our techniques give rise to attacks that can be efficiently applied with much shorter queries to the MAC oracle, and therefore are more widely applicable to concrete hash functions. In particular, we devise the first universal forgery attack applicable to HMAC with SHA-1 and SHA-2 (see Table 1).

1.2 Framework of the attacks

In order to recover an internal state computed by the MAC oracle during the processing of some message (namely, mount a state-recovery attack), we use a framework which is similar to the one of [19]. Namely, we match states that are computed offline with (unknown) states that are computed online (during the processing of messages by the MAC oracle). However, as arbitrary states

match with low probability (which does not lead to efficient attacks), we only match *special states*, which have a higher probability to be equal. These special states are the result of iterating random functions using chains, computed by applying the compression function on a fixed message from arbitrary initial states. In this paper, we exploit two types of special states which were also exploited in [19]: states on which two evaluated chains collide, and states on which a single chain collides with itself to form a cycle. We also introduce a third type of special states, which result from the reduction of the image space that occurs when applying a fixed sequence of random functions. This is used in some of our new attacks, and in particular against HAIFA.

As described above, after we compute special states both online and offline, we need to match them in order to recover an online state. However, since the online states are unknown, the matching cannot be performed directly, and we are forced to match the nodes indirectly using *filters*. A filter for a node (state) is a property that identifies it with high probability, *i.e.*, once the filters of two nodes match, then the nodes themselves match with high probability. Since the complexity of the matching steps in a state-recovery attack depends on the complexity of building a filter for a node and testing a filter on a node, we are interested in building filters efficiently. In this paper, we use two types of filters: collision filters (which were also used in [19]) and diamond filters, which exploit the diamond structure (introduced in [15]) in order to build filters for a large set of nodes with reduced average complexity. In fact, we use a novel online construction of the diamond structure via the MAC oracle, whereas such a structure is typically computed offline. In particular, we show that despite the fact that the online diamond filter increases the complexity of building the filter, the complexity of the actual matching phase is significantly reduced, and gives improved attacks in many cases.

1.2.1 Complexity evaluation

The complexity of our attacks is calculated in terms of the number of compression function evaluations of the underlying hash functions. Similarly to related papers in the field, we assume that sorting a table can be performed in linear time, while searching a sorted table takes constant time. As the complexities of all our attacks are exponential in the state size ℓ, we mostly use the big-O and soft-O ($\tilde{O}$) notation to estimate them (which is common practice in analysis of exponential-time algorithms). These estimations ignore small constants and polynomial factors in ℓ (which are generally linear in our attacks). An exception to this is the more precise complexity evaluation for the attack on `HMAC-Streebog`, given in Appendix B.

1.2.2 Outline

The paper is organized as follows. We begin with a description of `HMAC` in Section 2. We then describe and analyze the algorithms used to compute special states in Section 3, and the filters we use in our attacks in Section 4.

Next, we present a simple attack against HMAC with a HAIFA hash function in Section 5, and revisit the results of [19] in Section 6, presenting new tradeoffs for attacks on Merkle-Damgård hash functions. In Section 7, we give more complex attacks for shorter messages. Our key-recovery attack on HMAC with GOST R 34.11-2012 (Streebog) is described is Section 8. Finally, in Section 9, we present our universal forgery attacks with short queries, and conclude in Section 10.

2 HMAC and Hash-based MACs

In this paper we study MAC algorithms based on a hash function, such as HMAC. Using a hash function H, HMAC is defined as $\texttt{HMAC}(K, M) = H(K \oplus \texttt{opad} \,\|\, H(K \oplus \texttt{ipad} \,\|\, M))$, as shown in Figure 2. More generally, we consider a class of iterative designs based on a family of compression functions h_i and a finalization function g, represented by Figure 3. We denote the ℓ-bit internal state as x_i and the n-bit output tag as t. The message M is divided into p blocks m_i, and the MAC is computed as:

$$x_0 = I_K \qquad x_{i+1} = h_i(x_i, m_i) \qquad t = g(K, x_p, |M|).$$

The message processing updates the internal state starting from a key-dependant value I_K, and the output is produced with a key-dependant finalization function g. In particular, we note that the state update does not depend on the key. Our description covers HMAC [4], Sandwich-MAC [27] and envelope-MAC [26] with any common hash function. The hash function can use the message length in the finalization process, which is a common practice, and the round function can depend on a block counter, as in the HAIFA mode. If the hash function uses the plain Merkle-Damgård mode, the round functions h_i are all identical (this is the model analyzed in previous attacks [19,24]).

In this work, we assume that with very high probability, an arbitrary collision on the tag of two messages of the same length is a result of a collision on the final internal states x_p. This greatly simplifies the description of the attacks, and does not restrict the scope of our results. This assumption can be realized by altering the original scheme such that the new tag length is made larger than ℓ. Indeed, from a function $\mathrm{MAC}_1(K, M)$ with an output of n bits, we can build a function $\mathrm{MAC}_2(K, M)$ with a $2n$-bit output by appending message blocks $[0]$ and $[1]$ to M, as $\mathrm{MAC}_2(K, M) = \mathrm{MAC}_1(K, M \,\|\, [0]) \,\|\, \mathrm{MAC}_1(K, M \,\|\, [1])$. Our attacks applied to MAC_2 can immediately be turned to attacks on MAC_1 with a multiplicative penalty of 2.

In our attacks, we evaluate chains of the compression function h (or h_i for HAIFA) with a fixed message input block $[b]$ (usually $[b] = [0]$), and typically simplify our notation and define $f(x) = h(x, [b])$ (or $f_i(x) = h_i(x, [b])$ for HAIFA). We assume that the function f (or each f_i) is chosen from all ℓ-bit mappings uniformly at random. This implies that our analysis captures most (but not all) choices of underlying hash functions.

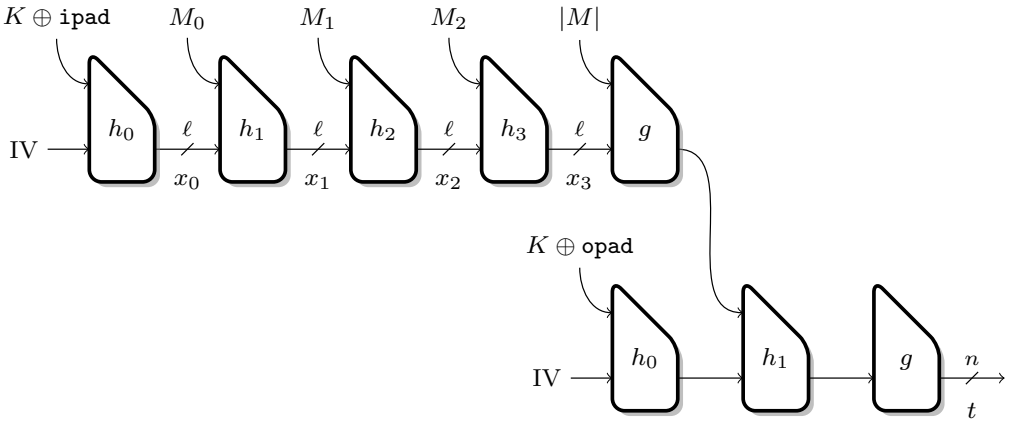


Fig. 2 `HMAC` with a HAIFA hash function. There are two hash function calls, each of them using the key at the beginning.

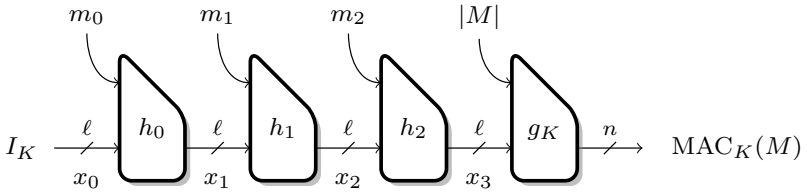


Fig. 3 Hash-based MAC with HAIFA. Only the initial value and the final transformation are keyed.

3 Description and Analysis of Collision Search Algorithms

In this section, we describe the collision search algorithms which are used in our state-recovery attacks in order to compute special states. We then analyze these algorithms (assuming they are applied to random functions) and prove the two conjectures of [19]. Lemma 1 proves the first conjecture, while Lemma 3 proves the second conjecture.

3.1 Collision search algorithms

We use standard collision search algorithms which evaluate chains starting from arbitrary points. Namely, a chain $\overrightarrow{x}$ starts from x_0, and is constructed iteratively by the equation $x_i = f_i(x_{i-1})$ up to $i = 2^s$ for a fixed value of s. We consider two different types of collisions between two chains $\overrightarrow{x}$ and $\overrightarrow{y}$: free-offset collisions ($x_i = y_j$ for any i, j, with all the f_i's being equal), and same-offset collisions ($x_i = y_i$).

3.1.1 Free-offset collision search

When searching offline for collisions in iterations of a *single random function* f, we evaluate 2^t chains starting from arbitrary points, and extended to length 2^s.

Assuming that $2^s \cdot 2^{t+s} \leq 2^\ell$ (*i.e.*, $2s+t \leq \ell$), then we are at (or below) the birthday paradox threshold and therefore each of the chains is not expected to collide with more than one other chain in the structure. This implies that the structure contains a total of about 2^{t+s} distinct points, and (according to the birthday paradox) we expect it to contain a total of $2^c = \Theta(2^{2(t+s)-\ell})$ collisions. We can easily recover all of these collisions in $O(2^{t+s}) = O(2^{(c+\ell)/2})$ time by storing all the evaluated points and checking for collisions in memory.

We note that we can reduce the memory requirements of the algorithm by using the parallel collision search algorithm of van Oorschot and Wiener [22]. However, in this paper, we generally focus on time complexity and do not try to optimize the memory complexity of our attacks.

3.1.2 Cycle search

Cycles are created when a chain collides with itself while iterating a single random function f. In order to search offline for a cycle of length $O(2^s)$ (for $s \leq \ell/2$), we evaluate $2^{\ell-2s}$ chains starting from arbitrary points, and extended to length 2^s. The probability that a chain collides with itself to form a cycle is equal (up to a constant factor) to the probability that its first half (of length 2^{s-1}) collides with its second half, which occurs with probability $\Theta(2^{2s-\ell})$. Thus, we expect to find a cycle within the evaluated $2^{\ell-2s}$ chains.

3.1.3 Same-offset collision search

While free-offset collisions are the most general form of collisions, they cannot always be efficiently detected and exploited by our attacks. In particular, they cannot be efficiently detected in queries to the online oracle (as a collision between messages of different lengths would lead to different values after the finalization function). Furthermore, if the hash function uses the HAIFA iteration mode, it is also not clear how to exploit free-offset collisions offline, as the colliding chains do not merge after the collision (and thus we do not have any easily detectable non-random property).

In the cases above, we are forced to only use collisions that occur at the same offset. When computing 2^t chains of length 2^s, a pair of chains collides at the same offset with probability of roughly $2^{s-\ell}$. As we have 2^{2t} pairs[2] of chains, we expect to find about $2^{2t+s-\ell}$ same-offset collisions.

We note that the computation above assumes that pairs of chains behave (almost) independently, which is the case when the total number of collisions

[2] More precisely, there are 2^{2t-1} unordered pairs of chains. However, our analysis is correct up to small constants.

in the structure of chains is relatively small. More precisely, the computation assumes that a constant fraction of the chains do not collide with any other chain, so that the chain structure contains $\Theta(2^{s+t})$ distinct points. When the iteration functions f_i are all equal, the condition reduces to $2s + t \leq \ell$ (as calculated for free-offset collision search). However, when the iteration functions f_i are distinct, we only have to consider collisions at the same offset. In order to collect the $2^{2t+s-\ell}$ same-offset collisions, we require that a constant fraction of the chains do not collide with any other chains, implying that $2t + s - \ell < t$, or $s + t \leq \ell$.

Locating collisions online. Online collisions are detected by sorting and comparing the tags obtained by querying the MAC oracle with chains of a fixed length 2^s. If we find two massages such that $\mathrm{MAC}(M) = \mathrm{MAC}(M')$, we can easily compute the message prefix that gives the (unknown) collision state, as described in [19]. Namely, if we denote by $M_{|i}$ the i-block prefix of M, then we find the smallest i such that $\mathrm{MAC}(M_{|i}) = \mathrm{MAC}(M'_{|i})$ using binary search. This algorithm queries the MAC oracle with $O(s)$ messages of length $O(2^s)$, and thus the time complexity of locating a collision online is $s \cdot 2^s = \tilde{O}(2^s)$.

3.2 Analysis of the collision search algorithms

In this section, we provide useful lemmas regarding the collision search algorithms described above. These lemmas are used in order to estimate the collision probability of special states that are calculated by our attacks and thus to bound their complexity. Lemmas 1, 2 and 5 can generally be considered as common knowledge in the field. Perhaps, the most interesting results in this section are lemmas 3 and 4. These lemmas show that the probability of our collision search algorithms to reach the same collision twice from different arbitrary starting points, is perhaps higher than one would expect. This phenomenon was already observed in previous works such as [22], but to the best of our knowledge, this is the first time that this phenomenon is formally proved.

Lemma 1 *Let $s \leq \ell/2$ be a non-negative integer. Let $f_1, f_2, \ldots, f_{2^s}$ be a sequence of random functions over the set of 2^ℓ elements, and $g_i \triangleq f_i \circ \ldots \circ f_2 \circ f_1$ (with the f_i being either all identical, or completely independent). Then, the images of two arbitrary inputs to g_{2^s} collide with probability of about $2^{s-\ell}$,* i.e. $\Pr_{x,y}\left[g_{2^s}(x) = g_{2^s}(y)\right] = \Theta(2^{s-\ell})$.

Proof Let x and y be two arbitrary points, $x_i = g_i(x)$ and $y_i = g_i(y)$ (or equivalently $x_0 = x,\ x_i = f_i(x_{i-1})$ and $y_0 = y,\ y_i = f_i(y_{i-1})$). We first deal with the case of independent functions f_i. As all f_i are random functions, we have a pair of random points (x_i, y_i) for each offset. The probability that none of the pairs collide is $(1 - 2^{-\ell})^{2^s}$. The event $g_{2^s}(x) = g_{2^s}(y)$ is equivalent to having (at least) one collision between the pairs, which occurs with probability $1 - (1 - 2^{-\ell})^{2^s} = \Theta(2^{s-\ell})$ (given $s \leq \ell$).

When the functions f_i are identical, the analysis is similar, but one also needs to consider dependencies between the sequences of points which are caused by collisions $x_i = y_j$ for $i \neq j$ (and also cycles of the form $x_i = x_j$). However, since $s \leq \ell/2$, we are below (or at) the birthday bound and therefore such collisions occur with at most a (small) constant probability, and their asymptotic contribution to the value of $\Pr_{x,y}[g_{2^s}(x) = g_{2^s}(y)]$ can be neglected. □

Lemma 2 *Let $s \leq \ell/2$ be a non-negative integer. Let f be a random function, then with high probability the image size of the function f^i is $O(2^\ell/i)$.*

Proof We will show that the expected image size of f^i is upper bounded by $2 \cdot 2^\ell/i$. Since the image size is positive, this upper bound proves the lemma, as for some $\alpha > 1$ the probability that the image size is more than $2\alpha \cdot 2^\ell/i$ is at most $1/\alpha$.

We calculate this expectation based on Theorem 2 of [9], which asserts that the expected image size of f^i is $(1-\tau_i)2^\ell$, where $\tau_0 = 0$ and $\tau_{i+1} = e^{-1+\tau_i}$. We show by induction on $i \geq 1$ that $\tau_i > 1-(2/i)$, which implies that the expected image size of f^i is less than $2 \cdot 2^\ell/i$, as required. The base is $i = 1$, for which indeed $\tau_1 = e^{-1} > -1$. Next, we need to prove that $\tau_{i+1} > 1-(2/(i+1))$, or $e^{-1+\tau_i} > 1-(2/(i+1))$. According to the assumption, it is sufficient to show that $e^{-2/i} > 1-(2/(i+1))$, or equivalently $-2/i > \ln(1-(2/(i+1)))$. This inequality can be deduced from the inequality $\ln(1+x) < x/(x/2+1)$ for $-1 < x < 0$ (see inequality (2) in [20]), by plugging in $x = -2/(i+1)$. □

Lemma 3 *Let $\hat{x}$ and $\hat{y}$ be two random collisions (same-offset or free-offset) found by a collision search algorithm using 2^t chains of length 2^s, with* ***a fixed ℓ-bit random function f*** *such that $2s + t \leq \ell$. Then $\Pr[\hat{x} = \hat{y}] = \Theta(2^{2s-\ell})$.*

Proof First, we note that we generally have 4 cases to analyze, according to whether $\hat{x}$ and $\hat{y}$ were found using a free-offset, or a same-offset collision search algorithm. However, the number of cases can be reduced to 3, as we have 2 symmetric cases, where one collision is free-offset, and the other is same-offset. In this proof, we assume that $\hat{x}$ is a same-offset collision and $\hat{y}$ is a free-offset collision (this is the configuration used in our attacks). However, the proof can easily be adapted to the 2 other settings.

As previously noted, when $2s + t \leq \ell$ each evaluated chain is not expected to collide with more than one different chain, and the pairs of chains can essentially be analyzed independently. Given a collision $\hat{x}$, We denote by A the event that 2 new chains of length 2^s, starting from arbitrary points (y_0, y_0'), also collide on $\hat{x}$. Below, we show that $\Pr[A] = \Theta(2^{2(2s-\ell)})$. Denote by B the event that the chains starting from (y_0, y_0') collide (on $\hat{x}$ or any other point), then $\Pr[B] = \Theta(2^{2s-\ell})$. We are interested in calculating the conditional probability $\Pr[A|B] = \Pr[\hat{x} = \hat{y}]$, and we have $\Pr[A|B] = \Pr[A \cap B]/\Pr[B] = \Pr[A]/\Pr[B] = \Theta(2^{2(2s-\ell)-(2s-\ell)}) = \Theta(2^{2s-\ell})$, as required.

We are left to show that $\Pr[A] = \Theta(2^{2(2s-\ell)})$. Denote the starting points of the chains which collide on $\hat{x}$ by (x_0, x_0'), and the actual corresponding

colliding points of the chains by (x_i, x'_i), so that $f(x_i) = f(x'_i) = \hat{x}$, with $x_i \neq x'_i$. Fixing (x_0, x'_0), we now calculate $\Pr[A]$, namely the probability that 2 new chains of length 2^s, starting from arbitrary points (y_0, y'_0), also collide on $\hat{x}$. This occurs if $y_0, y_1, \ldots, y_{2^s-i}$ collides with $x_0, x_1, \ldots, x_i$, and $y'_0, y'_1, \ldots, y'_{2^s-i}$ collides with $x'_0, x'_1, \ldots, x'_i$ (or vise-versa). Clearly, $\Pr[A] = O(2^{2(2s-\ell)})$, as all 4 chains are of length $O(2^s)$. Hence, to conclude the proof, we need to show that $\Pr[A] = \Omega(2^{2(2s-\ell)})$.

In order to simplify the proof, we assume in the following that $0.25 \cdot 2^s \leq i \leq 0.75 \cdot 2^s$ and denote this event by I. We have $\Pr[I] \approx 1/2$ since the offset of the collision $\hat{x}$ is roughly uniformly distributed in the interval $[0, 2^s]$. This can be shown using Lemma 1, as increasing the length of the chains increases the collision probability (at a common offset) by the same multiplicative factor.

We have $\Pr[A|I] = \Theta(2^{2(2s-\ell)})$, since when $0.25 \cdot 2^s \leq i \leq 0.75 \cdot 2^s$, all 4 chains $(y_0, y_1, \ldots, y_{2^s-i},\ x_0, x_1, \ldots, x_i,\ y'_0, y'_1, \ldots, y'_{2^s-i},\ x'_0, x'_1, \ldots, x'_i)$ are of length $\Theta(2^s)$. Therefore, $\Pr[A] = \Pr[A|I] \cdot \Pr[I] + \Pr[A|\neg I] \cdot \Pr[\neg I] \geq \Pr[A|I] \cdot \Pr[I] = \Omega(2^{2(2s-\ell)})$ as required. □

Lemma 4 *Let $\hat{x}$ and $\hat{y}$ be two arbitrary same-offset collisions found, respectively, at offsets i and j by a collision search algorithm using 2^t chains of fixed length 2^s, with **independent ℓ-bit random functions f_i**, such that $s + t < \ell$. Then $\Pr[(\hat{x}, i) = (\hat{y}, j)] = \Theta(2^{s-\ell})$. Furthermore, when $i = j$, we have $\Pr[\hat{x} = \hat{y}] = \Theta(2^{2s-\ell})$.*

Proof The proof follows essentially the same line of arguments as the proof of Lemma 3. We fix the collision $\hat{x}$ and denote by A the event that chains starting from arbitrary points (y_0, y'_0) collide on $\hat{x}$ at offset i, and by B the event that the chains starting from (y_0, y'_0) collide at an arbitrary offset j. We have $\Pr[B] = \Theta(2^{s-\ell})$ (see Lemma 1) and $\Pr[A] = \Theta(2^{2(s-\ell)})$.[3] Thus, $\Pr[A|B] = \Pr[A \cap B]/\Pr[B] = \Pr[A]/\Pr[B] = \Theta(2^{s-\ell})$ as claimed.

When assuming that $i = j$, we need to change the definition of event B such that the chains starting from (y_0, y'_0) collide at the fixed offset i. This gives $\Pr[B] = \Theta(2^{-\ell})$ and $\Pr[A|B] = \Pr[A \cap B]/\Pr[B] = \Pr[A]/\Pr[B] = \Theta(2^{2s-\ell})$. □

Lemma 5 *Let $\hat{x}$ be the entry point of an arbitrary cycle found by the cycle search algorithm for a fixed ℓ-bit function f, using chains of fixed length 2^s such that $s \leq \ell/2$. Let y_0 be an arbitrary point, and define the chain $y_{i+1} = f(y_i)$ for $i \in \{0, 1, \ldots, 2^s - 1\}$. Then the probability that the chain y_i finds the same cycle and entry point $\hat{x}$ is $\Theta(2^{2s-\ell})$,* i.e. *$\Pr[\exists i', j' \mid y_{i'} \neq y_{j'}$ and $y_{i'+1} = y_{j'+1} = \hat{x}] = \Theta(2^{2s-\ell})$.*

Proof We denote the starting point of the chain found by the cycle search algorithm which collides (cycles) on $\hat{x}$ by x_0, and the corresponding collision by (x_i, x_j), with $i < j$, $x_i \neq x_j$ and $f(x_i) = f(x_j) = \hat{x}$. In the following, we assume that $0.25 \cdot 2^s \leq i, j \leq 0.75 \cdot 2^s$, which occurs with constant

[3] Once again, a lower bound on $\Pr[A]$ is easy to calculate assuming that $0.25 \cdot 2^s \leq i \leq 0.75 \cdot 2^s$ (which occurs with probability of about 1/2).

probability. The chain $\overrightarrow{y}$ will cycle with entry point $\hat{x}$ if $y_0, y_1, \ldots, y_{2^s-j}$ collides with $x_0, x_1, \ldots, x_i$. This occurs with probability of about $2^{2s-\ell}$ (when $0.25 \cdot 2^s \leq i, j \leq 0.75 \cdot 2^s$, the two chains are of length $\Theta(2^s)$), which proves that $\Pr[\exists i', j' \mid y_{i'} \neq y_{j'} \text{ and } y_{i'+1} = y_{j'+1} = \hat{x}] = \Omega(2^{2s-\ell})$. On the other hand, the probability that two chains of length at most 2^s collide is at most (roughly) $2^{2s-\ell}$, and therefore $\Pr[\exists i', j' \mid y_{i'} \neq y_{j'} \text{ and } y_{i'+1} = y_{j'+1} = \hat{x}] = \Theta(2^{2s-\ell})$. □

4 Filters

We describe the two types of filters that we use in our attacks in order to match (known) states computed offline with unknown states computed online by the MAC oracle.

4.1 Collision filters

A simple collision filter was introduced in [19], and is also used in our work. A collision filter for an offline state x is a pair of message blocks (b, b'), with $b \neq b'$, such that we obtain the same state after processing these blocks from x (*i.e.* $h(x, b) = h(x, b')$). In order to build the filter, we find a collision in the underlying hash function by evaluating its compression function with state x, and about $2^{\ell/2}$ different messages blocks b. We apply the filter online on an unknown node x' — obtained after processing a message m' — by checking whether the tags of $m' \parallel b$ and $m' \parallel b'$ collide. If the state obtained after processing m' is not x, the tags of $m' \parallel b$ and $m' \parallel b'$ collide with probability only $2^{-n} < 2^{-\ell}$; therefore the collision filter identifies the state x with high probability.

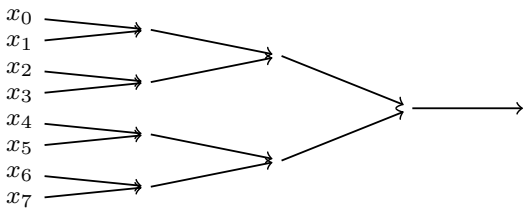
The complexity of building a collision filter offline is $O(2^{\ell/2})$. Testing the filter online requires querying the MAC oracle with $m' \parallel b$ and $m' \parallel b'$; assuming that the length of m' is $2^{s'}$, this requires $O(2^{s'})$ time.

4.2 Diamond filters

In order to build filters for 2^t nodes, we can build a collision filter for each one of them separately, requiring a total of $O(2^{t+\ell/2})$ time. However, this process can be optimized using the diamond structure, introduced by Kelsey and Kohno in the herding attack [15]. We now recall the details of this construction.

The diamond structure is built from a set of 2^t states x_i, constructing a set of messages m_i of length $O(t)$, such that iterating the compression function from any state x_i using message m_i leads to a fixed final state y. The structure is built in $O(t)$ iterations, where each iteration processes a layer of nodes and outputs a smaller layer to be processed by the next iteration. This process terminates once the layer contains only one node, which is denoted by y.

Starting from the first layer with 2^t points, we evaluate the compression function from each point x_i with about $2^{(\ell-t)/2}$ random message blocks. This gives a total of about $2^{(\ell+t)/2}$ random values, and we expect them to contain about 2^t collisions. Each collision allows matching two different values x_i, x_j and to send them to a common value in the next layer, such that its size is reduced by a factor about 2. The message m_i for a state x_i is constructed by concatenating the $O(t)$ message blocks on its path leading to y. According to the detailed analysis of [17], the time complexity of building the structure is $\Theta(2^{(\ell+t)/2})$. Note that for a HAIFA hash function, the nodes of each layer in the structure must be built using the same function and therefore they must have the same offset.



Once we finish building the diamond structure, we construct a standard collision filter for the final node y, using message blocks $([b], [b'])$. Thus, building a diamond filter offline for 2^t states requires $O(2^{(\ell+t)/2})$ time, which is faster than the $O(2^{t+\ell/2})$ time required to build a collision filter for each node separately.

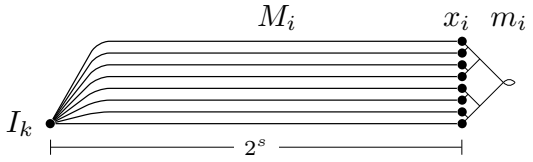
In order to test the filter for a state x_i (in the first layer of the diamond structure) on the unknown node x' obtained after processing a message m' online, we simply check whether the tags of $m' \parallel m_i \parallel [b]$ and $m' \parallel m_i \parallel [b']$ collide. Assuming that the length of m' is $2^{s'}$, then the online test requires $O(t + 2^{s'})$ time. Note that for a HAIFA hash function, the online and offline nodes tested for equality must have the same offset.

4.2.1 Online diamond filter

A novel observation that we use in this paper, is that in some attacks it is more efficient to build the diamond structure online by calling the MAC oracle. Namely, we construct a diamond structure for the set of 2^t states x_i, where (the unknown) x_i is a result of querying the MAC oracle with a message M_i. Note that the online construction is indeed possible, as the construction algorithm does not explicitly require the value of x_i, but rather builds the corresponding m_i by testing for collisions between the states (which can be detected according to collisions in the corresponding tags). However, testing for collisions online requires that all the messages M_i for which we build the online diamond filter are of the same length (both for HAIFA and Merkle-Damgård hash functions). Assuming that the messages M_i are of length 2^s, building this construction requires $O(2^{s+(t+\ell)/2})$ calls to the compression function.

In order to test the filter for an unknown online state x_i on a known state x', we simply evaluate the compression function from x' on $m_i \parallel [b]$ and $m_i \parallel [b']$,

and check whether the resulting two states are equal. Thus, the offline test requires $O(t)$ time.



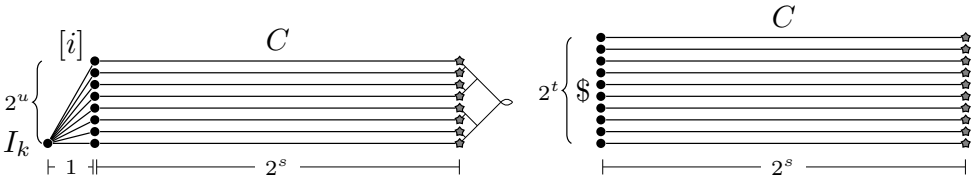
5 Internal State-recovery for HMAC with HAIFA

In this section, we describe the first internal state-recovery attack applicable to HAIFA. Our attack has a complexity of $\tilde{O}(2^{\ell-s})$ using messages of length 2^s, but this only applies with $s \leq \ell/5$; the lowest complexity we can reach is roughly $2^{4\ell/5}$. We note that attacks against HAIFA can also be used to attack a Merkle-Damgård hash function, giving more freedom in the queried messages by removing the requirement for long sequences of identical blocks as in [19].

In this attack, we fix a long sequence of "random" functions in order to reduce the entropy of the image states, based on Lemma 1. We then use an online diamond structure to match the states computed online with states that are compute offline. The detailed attack is as follows:

Attack 1: State-recovery attack against HMAC with HAIFA
Complexity $\tilde{O}(2^{\ell-s})$, with $s \leq \ell/5$ (min: $2^{4\ell/5}$)

1. (online) Fix a message C of length 2^s. Query the oracle with 2^u messages $M_i = [i] \parallel C$. Build an online diamond filter for the set of unknown states X, obtained after M_i.
2. (offline) Starting from 2^t arbitrary starting points, iterate the compression function with the fixed message C.
3. (offline) Test each image point x' obtained in Step 2 against each of the unknown states of X. If a match is found, then with high probability the state reached after the corresponding M_i is x'.



We detect a match between the grey points (⋆) using the diamond test built online.

Complexity analysis. In Step 3, we match the set X of size 2^u (implicitly computed during Step 1), and a set of size 2^t (computed during Step 2). We compare 2^{t+u} pairs of points, and each pair collides with probability $2^{s-\ell}$

according to Lemma 1. Therefore, the attack is successful with high probability if $t+u \geq \ell - s$. We now assume that $t = \ell - s - u$, and evaluate the complexity of each step of the attack:

Step 1: $2^{s+u/2+\ell/2}$ **Step 2:** $2^{s+t} = 2^{\ell-u}$ **Step 3:** $2^{t+u} \cdot u = 2^{\ell-s} \cdot u$

The lowest complexity is reached when all the steps of the attack have the same complexity with $s = u = \ell/5$. More generally, we assume that $s \leq \ell/5$ and set $u = s$ to balance steps 2 and 3. This gives an attack with complexity $\tilde{O}(2^{\ell-s})$ since $s + u/2 + \ell/2 = 3s/2 + \ell/2 \leq 4\ell/5 \leq \ell - s$.

6 New Tradeoffs for Merkle-Damgård

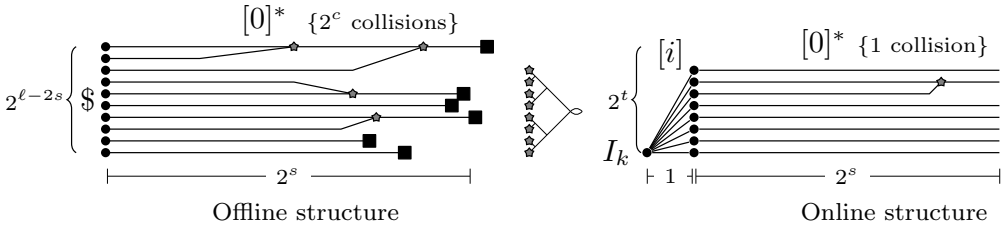
In this section, we revisit the results of [19], and give more flexible tradeoffs for various message lengths.

6.1 Tradeoff based on iteration chains

In this attack, we match special states obtained using collision based on Lemma 3. This attack extends the original tradeoff of [19] by using two improved techniques. First, while [19] used a same-offset offline collision search, we use a more general free-offset offline collision search which enables us to find collisions more efficiently. Secondly, while [19] used collision filters, we use a more efficient diamond filter.

Attack 2: Chain-based tradeoff for `HMAC` with Merkle-Damgård
Complexity $O(2^{\ell-s})$, with $s \leq \ell/3$ (min: $2^{2\ell/3}$)

1. (offline) Use free-offset collision search from $2^{\ell-2s}$ starting points with chains of length 2^s, and find 2^c collisions (denoted by the set $\hat{X}$).
2. (offline) Build a diamond filter for the points in $\hat{X}$.
3. (online) Query the oracle with 2^t messages $M_i = [i] \parallel [0]^{2^s}$. Sort the tags, and locate 1 collision among the tags.
4. (online) Use a binary search to find the message prefix giving the unknown online collision state $\hat{y}$.
5. (online) Match the unknown online state $\hat{y}$ with each offline state in $\hat{X}$ using the diamond filter. If a match with $\hat{x} \in \hat{X}$ is found, then with very high probability $\hat{y} = \hat{x}$.



We generate collisions offline using free-offset collision search, build a diamond filter for the collision points ($\star$), and recover the state of an online collision.

Complexity analysis. In Step 1, we use free-offset collision search with $2^{\ell-2s}$ starting points and chains of length 2^s, and thus according to Section 3.1, we find $2^{\ell-2s}$ collisions (*i.e.* $c = \ell - 2s$). Furthermore, according to Lemma 3, $\hat{y} \in \hat{X}$ with high probability, in which case the attack succeeds.

In Step 3, we use same-offset collision search with 2^t starting points and chains of length 2^s, and thus according to Section 3.1, we find $2^{2t+s-\ell}$ collisions. As we require one collision, we have $t = (\ell - s)/2$. We now compute the complexity of each step of the attack:

Step 1:	$2^{\ell-s}$	**Step 2:**	$2^{\ell/2+c/2} = 2^{\ell-s}$
Step 3:	$2^{t+s} = 2^{(\ell+s)/2}$	**Step 4:**	$s \cdot 2^s$
Step 5:	$2^{c+s} = 2^{\ell-s}$		

With $s \leq \ell/3$, we have $(\ell + s)/2 \leq 2/3 \cdot \ell \leq \ell - s$, and the complexity of the attack is $O(2^{\ell-s})$.

6.2 Tradeoff based on cycles

We now generalize the cycle-based state-recovery attack of [19] which exploits the main cycle of approximate length $2^{\ell/2}$ in the graph of the random mapping in order to construct two colliding messages of the same length (thus having equal tags, which can be detected at the output). The attack of [19] uses messages of length $2^{\ell/2}$ and has a complexity of $2^{\ell/2}$.

Our attack uses the same idea of [19], but searches for (potentially) shorter cycles using (potentially) shorter messages of length 2^s for $s \leq \ell/2$. The complexity of the attack $2^{2\ell-3s}$.

Attack 3: Cycle-based tradeoff for `HMAC` with Merkle-Damgård
Complexity $O(2^{2\ell-3s})$, with $s \leq \ell/2$ (min: $2^{\ell/2}$)

1. (offline) Search for a cycle in the functional graph of $f = h_{[0]}$, using the algorithm of Section 3.1 with chains of length 2^s. Denote the length of the cycle by L, and its entry point by $\hat{x}$. Build a collision filter for $\hat{x}$.
2. (online) For different values of the message block $[b]$, query the MAC oracle with two messages $M = [b] \parallel [0]^{2^s} \parallel [1] \parallel [0]^{2^s+L}$ and $M' =$

$[b] \| [0]^{2^s+L} \| [1] \| [0]^{2^s}$ (both of length $1+2^s+1+2^s+L = 2+2^{s+1}+L$), until $\mathrm{MAC}(M) = \mathrm{MAC}(M')$.

3. (online) Define $M_i = [b] \| [0]^i \| [1] \| [0]^{2^s+L}$ and $M'_i = [b] \| [0]^{i+L} \| [1] \| [0]^{2^s}$, and use a binary search to find the minimum value of i such that M_i and M'_i have the same MAC. Use the collision filter to test whether the state reached after $[b] \| [0]^i$ is equal to $\hat{x}$.

Complexity analysis. The intuition behind this attack is the same as the cycle-based attack from [19]. Step 2 will be successful if the message $M_0 = [b] \| [0]^{2^s} \| [1] \| [0]^{2^s}$ reaches the cycle of length L twice: once after processing $[b] \| [0]^{2^s}$ and again after processing the full message M'. Indeed, if this is the case, then adding L zero blocks in the first cycle (in message M') or in the second cycle (in message M) does not change the state reached before the finalization function. In addition, M and M' have the same length $2+2^{s+1}+L$ which allows the collision to be propagated to the output (a simple attack that enters the cycle only once would not work because of the different lengths).

More formally, we need the two following conditions to be satisfied. First, the states obtained after evaluating the prefixes $[b] \| [0]^{2^s}$ and $[b] \| [0]^{2^s+L}$ must collide. This occurs if one of the states computed during the evaluation of $[b] \| [0]^{2^s}$ collides with $\hat{x}$ (and thus enters the cycle of length L), which has probability $\Theta(2^{2s-\ell})$ according to Lemma 5. Secondly, the states obtained after evaluating the suffixes $[1] \| [0]^{2^s+L}$ and $[1] \| [0]^{2^s}$ must also collide. Similarly to the previous case, this occurs if one of the states computed during the evaluation of $[1] \| [0]^{2^s}$ collides with $\hat{x}$. Again, this event occurs with probability $\Theta(2^{2s-\ell})$ according to Lemma 5. Thus, the success probability of Step 2 is $\Omega(2^{2(2s-\ell)})$ (if fact, it is $\Theta(2^{2(2s-\ell)})$), and we need to repeat it for $O(2^{2(\ell-2s)})$ different values of $[b]$ for the attack to succeed with high probability. Consequently, the time complexity of Step 2 is $O(2^{2(\ell-2s)+s}) = O(2^{2\ell-3s})$.

Step 3 will detect the minimum i such that M_i reaches the cycle, therefore the i-th state is the entry point of the cycle. Step 3 is expected to be run once for the correct pair, but each block value b can lead to an internal collision (at a common offset) without reaching the cycle with probability about $2^{s-\ell}$, according to Lemma 1. Thus, Step 3 is run $2^{2(\ell-2s)+s-\ell} = 2^{\ell-3s}$ times with false positives.

The time complexity of all the steps is summarized below.

Step 1: $2^{\ell-2s+s} = 2^{\ell-s}$ **Step 2:** $2^{2\cdot(\ell-2s)+s} = 2^{2\ell-3s}$

Step 3: $2^s \cdot s \cdot (1 + 2^{\ell-3s}) = (2^s + 2^{\ell-2s}) \cdot s$

Since $s \leq \ell/2$, the complexity of the attack is $O(2^{2\ell-3s})$.

7 Shorter Message Attacks

In this section, we describe more complex attacks that can reach a tradeoff of $2^{\ell-2s}$ for relatively small values of s. These attacks are useful in cases where the

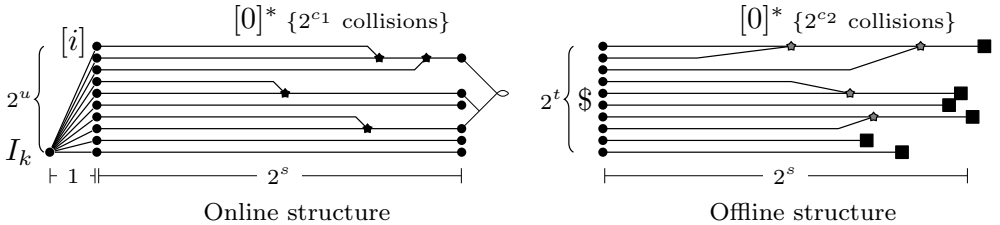
message length of the underlying hash function is very restricted (*e.g.* the SHA-2 family), and moreover they play an important role in the key-recovery attack on HMAC with the GOST R 34.11-2012 hash function (described in Section 8). In order to reach a complexity of $2^{\ell-2s}$, we combine the idea of building filters in the online phase with the use of collisions as special states (using the results of Lemma 3 for Merkle-Damgård and Lemma 4 for HAIFA).

In the case of Merkle-Damgård with identical compression functions, we reach a complexity of $2^{\ell-2s}$ for $s \leq \ell/8$, *i.e.* the optimal complexity of this attack is $2^{3/4 \cdot \ell}$. With the HAIFA mode of operation, we reach a complexity of $2^{\ell-2s}$ for $s \leq \ell/10$ *i.e.* the optimal complexity of $2^{4/5 \cdot \ell}$, matching the optimal complexity of the attack of Section 5.

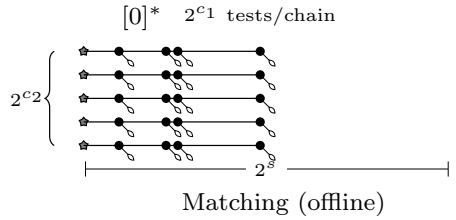
7.1 Merkle-Damgård

Attack 4: Short message attack for HMAC with Merkle-Damgård
Complexity $\tilde{O}(2^{\ell-2s})$, with $s \leq \ell/8$ (min: $2^{3\ell/4}$)

1. (online) Query the oracle with 2^u messages $M_i = [i] \parallel [0]^{2^s}$, and locate 2^{c_1} collisions.
2. (online) For each collision (i, j), use a binary search to find the distance (offset) d_{ij} from the starting point to the collision, and denote the (unknown) state reached after M_i (or M_j) by y_{ij}.
   Denote the set of all y_{ij} (containing about 2^{c_1} states) by Y. Build an online diamond filter for all the states in Y.
3. (offline) Run a free-offset collision search algorithm from 2^t starting points with chains of length 2^s, and locate 2^{c_2} collisions.
4. (offline) For each offline collision $\hat{x}$, match its iterates with all points $y_{ij} \in Y$: iterate the compression function with a zero message starting from $\hat{x}$ (up to 2^s times), and match iterate $2^s - d_{ij}$ (*i.e.*, $f^{2^s - d_{ij}}(\hat{x})$) with y_{ij} using the diamond filter. If a match is found, then with high probability $y_{ij} = f^{2^s - d_{ij}}(\hat{x})$.



We generate collisions, build an online diamond filter for their endpoints, and match them with iterates of collisions found offline.



Complexity analysis. Using similar analysis to Section 6.1, we have $c_1 = 2u + s - \ell$ (as a pair of chains collides at the same offset with probability $2^{s-\ell}$, and we have 2^{2u} such pairs) and $c_2 = 2t + 2s - \ell$. The attack succeeds if the sets of collisions found online and offline intersect. According to Lemma 3, this occurs with high probability if $c_1 + c_2 \geq \ell - 2s$. In the following, we assume $c_1 + c_2 = \ell - 2s$.

Step 1: $2^{u+s} = 2^{s/2+c_1/2+\ell/2}$ **Step 2:** $2^{s+c_1/2+\ell/2} = 2^{\ell-c_2/2}$

Step 3: $2^{t+s} = 2^{\ell/2+c_2/2}$ **Step 4:** $2^{c_2+s} + 2^{c_1+c_2} \cdot c_1 = 2^{c_2+s} + 2^{\ell-2s} \cdot c_1$

The best tradeoffs are achieved by balancing steps 2 and 3, *i.e.* with $c_2 = \ell/2$, implying that $c_1 = \ell/2 - 2s$. This reduces the complexity to:

Step 1: $2^{3\ell/4-s/2}$ **Step 2:** $2^{3\ell/4}$

Step 3: $2^{3\ell/4}$ **Step 4:** $2^{\ell/2+s} + 2^{\ell-2s} \cdot \ell/2$

With $s \leq \ell/8$, we have $\ell/2 + s \leq 5\ell/8$ and $3\ell/4 \leq \ell - 2s$; therefore the complexity of the attack is $\tilde{O}(2^{\ell-2s})$.

We note that the complexity can be reduced to a minimum of $O(2^{3\ell/4})$ (without logarithmic factors) by using messages which are slightly longer than $2^{\ell/8}$. In particular, this gives optimal attacks for functions of the `SHA-2` family.

7.2 HAIFA

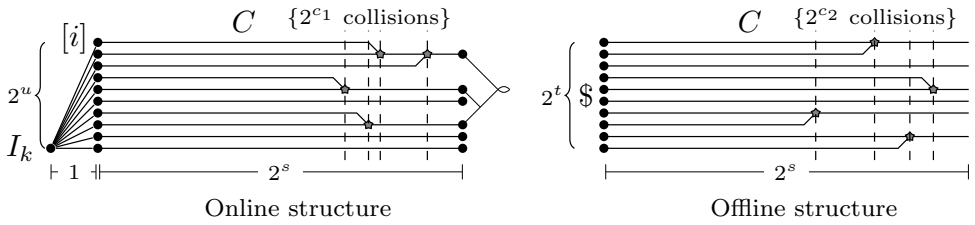
The general structure of the attack on HAIFA is similar to one of the previous attack on Merkle-Damgård. The main difference between the attacks is that we are forced to use same-offset collision search offline, rather than free-offset collision search as in the previous attack.

Attack 5: Short message attack for `HMAC` with HAIFA
Complexity $\tilde{O}(2^{\ell-2s})$, with $s \leq \ell/10$ (min: $2^{4\ell/5}$)

1. (online) Fix an arbitrary message suffix C of length 2^s, query the oracle with 2^u messages $M_i = [i] \,\|\, C$, and locate 2^{c_1} collisions.
2. (online) For each collision (i, j), use a binary search to find the distance (offset) d_{ij} from the starting point to the collision, and denote the (unknown) state reached after M_i (or M_j) by y_{ij}.

Denote the set of all y_{ij} (containing about 2^{c_1} states) by Y. Build an online diamond filter for all the states in Y.

3. (offline) Run a same-offset collision search algorithm by computing the compression function with the message C from 2^t arbitrary starting points, and locate 2^{c_2} collisions.
4. (offline) For each offline collision $\hat{x}$, match the endpoint of its chain with all online collisions $y_{ij} \in Y$ that occur at the same offset as $\hat{x}$. More precisely, for each $\hat{x}$ with offset d, use the diamond filter to test only the endpoint of its chain (at offset 2^s) with the corresponding collisions in Y that occur at offset $d_{ij} = d$. If a match with some y_{ij} is found, then with high probability the state reached after M_i and M_j is the endpoint of the offline collision $\hat{x}$.



We generate collisions and build an online diamond filter, and match the endpoints with offline collisions endpoints using the collision offset as a first filter.

Analysis. The attack succeeds in case there is a match between the set of collisions detected online and offline that occur at the same offset. According to Lemma 4, this match occurs with high probability when $c_1 + c_2 \geq \ell - s$, and thus we assume that $c_1 + c_2 = \ell - s$.

Complexity analysis. Similarly to the analysis of the previous attack on Merkle-Damgård, we have $c_1 = 2u + s - \ell$, but as we use same-offset collision search offline, we can detect a smaller number of $c_2 = 2t + s - \ell$ collisions. We note that the online collision offsets d_{ij} are essentially uniform (see Lemma 1), and therefore in Step 4, each offline collision $\hat{x}$ at offset d is matched with about $2^{c_1} \cdot 2^{-s}$ states in Y.

Step 1:	$2^{u+s} = 2^{s/2+c_1/2+\ell/2}$	**Step 2:**	$2^{s+c_1/2+\ell/2} = 2^{\ell-c_2/2+s/2}$
Step 3:	$2^{s+t} = 2^{s/2+c_2/2+\ell/2}$	**Step 4:**	$2^{c_1+c_2-s} \cdot c_1 = 2^{\ell-2s} \cdot c_1$

The optimal tradeoffs are achieved by balancing steps 2 and 3, *i.e.* with $c_2 = \ell/2$, implying that $c_1 = \ell/2 - s$. This reduces the complexity to:

Step 1:	$2^{3\ell/4}$	**Step 2:**	$2^{3\ell/4+s/2}$
Step 3:	$2^{3\ell/4+s/2}$	**Step 4:**	$2^{\ell-2s} \cdot \ell/2$

With $s \leq \ell/10$, we have $3\ell/4 + s/2 \leq 4\ell/5 \leq \ell - 2s$; therefore the complexity of the attack is $\tilde{O}(2^{\ell-2s})$.

8 Key-Recovery Attack on HMAC with GOST R 34.11-2012 (Streebog)

In this section we are interested in hash functions that use an internal checksum, as shown in Figure 4. In [19], the state-recovery attack on HMAC with a Merkle-Damgård hash function was extended into a key-recovery attack, in case the hash function uses an internal checksum like the GOST R 34.11-94 hash function. Here, we show that a similar attack can be applied to a hash function based on HAIFA with an internal checksum. Namely, the state-recovery attack (with complexity $2^{4\ell/5}$) can be extended into a key-recovery attack (with complexity $2^{4\ell/5}$).

In particular, this attack is applicable to the standard GOST R 34.11-2012 (aka Streebog), and gives a key-recovery attack with complexity 2^{417} for its 512-bit version. This result shows that HMAC-GOST R 34.11-2012 is weaker than HMAC-SHA-512 (for which no key-recovery attack better than exhaustive search is known).

8.1 Description of the attack

The attack uses the same framework as [19], exploiting the structure of hash functions with a checksum. We target the finalization function in the first hash function call. The input state value $x_\star$ can be recovered using the previous state-recovery attacks, and we exploit the fact that the checksum value is key dependant, but can be controlled by injecting differences in the message: $\sigma = K \oplus \texttt{ipad} \oplus \texttt{Sum}^{\oplus}(M)$. This allows for attacks which are somewhat similar to related-key attacks.

More precisely, we first generate a large set of messages of length L, leading to the same state $x_\star$, but with different checksums σ. Then, we consider collisions in the function $\sigma \mapsto g(x_\star, L, \sigma)$, which can be detected using online calls to the MAC oracle. At the same time, we can also generate such collisions offline since $x_\star$ and L are known. Moreover, if we find two collisions with the same difference in the σ input, there is a high probability that the actual σ values are the same, because the other inputs to g ($x_\star$ and L) are fixed (on average, we expect a single collision with a fixed difference). Once we find an online collision and an offline collision with the same difference, we can therefore recover the value of K by exploiting the equation $\sigma = K \oplus \texttt{ipad} \oplus \texttt{Sum}^{\oplus}(M)$.

8.2 Detailed attack process

The first step of the attack uses the state-recovery attack of Section 7.2. However, unlike hash functions without checksums, equal tags of arbitrary messages of the same length (but with different checksums) do not imply equality (with high probability) of the states reached after the messages are processed. Therefore, we execute Attack 5 with the modifications below that ensure we only compare tags of messages with a fixed checksum.

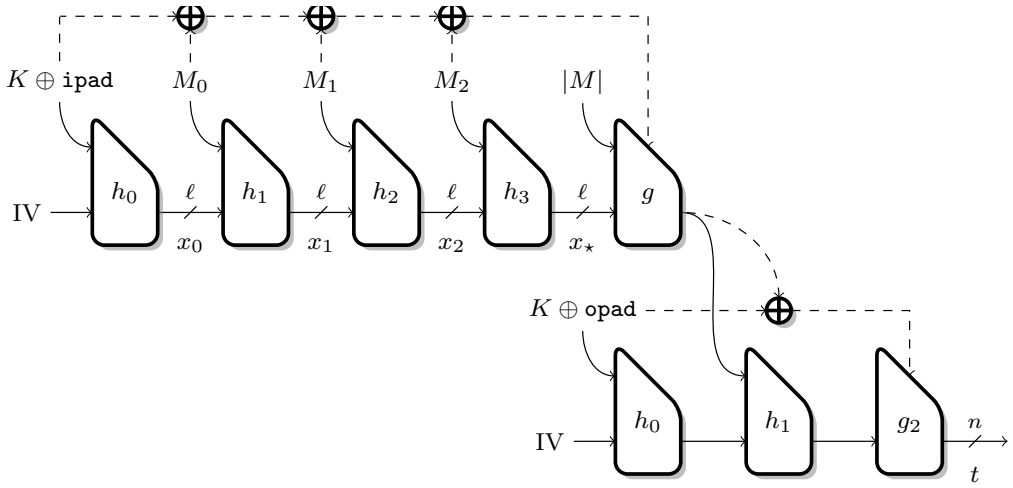


Fig. 4 HMAC based on a hash function with a block counter and a checksum (dashed lines).

- In step 1, we use $M_i = [i] \parallel [i] \parallel C$
- In step 2, when building the diamond structure, we extend the messages by $m \parallel m$. We do the same when building a collision pair for the endpoint of the diamond.
- For the offline steps, we ignore the checksum, and only look for collisions in the iterated state.

Attack 6: Key-recovery attack against HMAC with a GOST-like hash function
Complexity $\tilde{O}(2^{4\ell/5})$

0. Execute Attack 5 to recover a state x_1 obtained after processing some message M_1, with $|M_1| = s = \ell/10$.
1. (offline) Starting from state x_1, use Joux's multicollision attack [14] to generate a set of $2^{7\ell/10}$ messages that all collide on an internal state before the checksum block, but with different checksums. Denote the final state as $x_\star$, and the length of the messages as L.
2. (online) Query the MAC oracle with the set of messages from Step 1 and collect collisions ($2^{2\cdot 7\ell/10-\ell} = 2^{2\ell/5}$ collisions are expected). For each collision (M, M'), compute the checksum difference $\Delta M = \mathtt{Sum}^{\oplus}(M) \oplus \mathtt{Sum}^{\oplus}(M')$, and store $(\Delta M, \mathtt{Sum}^{\oplus}(M))$.
3. (offline) Choose a set of $2^{4\ell/5}$ one-block random checksums σ, compute $g(x_\star, L, \sigma)$ and collect collisions ($2^{3\ell/5}$ collisions are expected). For each collision (σ, σ'), compute the difference $\sigma \oplus \sigma'$ and compare it with the stored ΔM from Step 2. If a match is found, consider $\mathtt{Sum}^{\oplus}(M) \oplus \sigma \oplus$ ipad and $\mathtt{Sum}^{\oplus}(M) \oplus \sigma' \oplus$ ipad as potential key candidates, and test them using a known tag.

Analysis. Since we have $2^{2\ell/5}$ collisions in Step 2 and $2^{3\ell/5}$ collisions in Step 3, there is a high probability to find a match which is likely to reveal the key.

Complexity.

Step 0:	$\tilde{O}(2^{4\ell/5})$	**Step 1:**	$\ell \cdot 2^{\ell/2}$
Step 2:	$2^{s+7\ell/10} = 2^{4\ell/5}$	**Step 3:**	$2^{4\ell/5}$

The total complexity is $\tilde{O}(2^{4\ell/5})$. In Appendix B, we give a more precise complexity evaluation of this attack and show that it requires less than 2^{417} operations on average when applied to `Streebog` (with $\ell = 512$).

Significance of short message attacks. The efficient state-recovery attack for short messages is an important factor here, as the message length directly influences the time complexity of the attack in Step 2. In fact, state-recovery attacks with complexity $2^{\ell-s}$ as in Section 5 (or in [19], for the Merkle-Damgård construction) can only reach complexity $2^{5\ell/6}$ for key-recovery, while the attacks with complexity $2^{\ell-2s}$ (described in Section 7) allow to reach complexity $2^{4\ell/5}$.

We note that in `GOST` R 34.11-94 (which is based on Merkle-Damgård as opposed to HAIFA) the message length is processed with the same function as the other blocks. This (in addition to some padding properties) allowed the attack of [19] to deduce the state of a short message from the state of a long message. Therefore, unlike our key-recovery attack on `HMAC` with `GOST` R 34.11-2012, efficient state-recovery attacks for short messages did not play an important role in the key-recovery attack on `HMAC` with `GOST` R 34.11-94 [19].

9 Universal Forgery Attacks with Short Queries

In the setting of universal forgery attacks the adversary receives a challenge message C at the beginning of the game. The goal of the adversary is to predict the tag of the challenge by interacting with the MAC oracle, but without querying for the actual challenge C.

We now revisit the universal forgery attack of Peyrin and Wang [24]. This attack has two phases, where in the first phase the adversary recovers one of the internal states computed by the MAC oracle on the challenge C. In the second phase, the adversary uses a second-preimage attack on long messages in order to generate a different message C' known to have the same tag as the challenge. Thus, by querying the MAC oracle for C', the adversary can forge the tag of C. The first phase of the attack is the most expensive; the attack of [24] requires $\tilde{O}(2^{5\ell/6})$ computations, which has been improved to $\tilde{O}(2^{3\ell/4})$ in [11] using a heuristic assumption on the functional graph of random functions.

The main drawback of those attacks is that its first phase uses very long queries to the MAC oracle, regardless of the length of the challenge.[4] Therefore,

[4] The first phase of the attacks of [11,24] uses queries of length $2^{\ell/2}$. Their algorithms are related to the cycle-based attack of [19] that is generalized is Section 6.2.

the attack is inapplicable to many concrete HMAC instantiations where the hash function limits the message length (such as HMAC-SHA-1 and HMAC-SHA-2).

In this section, we use the tools developed in this paper to devise two universal forgery attacks using shorter queries to the MAC oracle. For a challenge of length 2^s, our first universal forgery attack has a complexity of $\tilde{O}(2^{\ell-s})$ with $s \leq \ell/7$, using queries to the MAC oracle of length of at most 2^{2s} (which is significantly smaller than $2^{\ell/2}$ for any $s \leq \ell/7$). Thus, the optimal complexity of this attack is $\tilde{O}(2^{6\ell/7})$, obtained with a challenge of length at least $2^{\ell/7}$ and queries of length $2^{2\ell/7}$. Our second universal forgery attack has a complexity of only $\tilde{O}(2^{\ell-s/2})$. However, it is applicable for any $s \leq 2\ell/5$, using queries to the MAC oracle of length of at most 2^s. Thus, this attack has an improved optimal complexity of $\tilde{O}(2^{4\ell/5})$, which is obtained with a challenge of length at least $2^{2\ell/5}$ and queries of length $2^{2\ell/5}$.

In order to devise our attacks, we construct new state-recovery algorithms, but reuse the second phase from [24] (*i.e.*, the second-preimage attack) in both of the attacks. Thus, in the following, we concentrate on the state-recovery algorithms. For the sake of completeness, we describe the second phase of the attack in Appendix A. Since its complexity is about $2^{\ell-s}$ for any $s \leq \ell/2$, it does not add a significant factor to the total time complexity.

9.1 A universal forgery attack based on the reduction of the image-set size

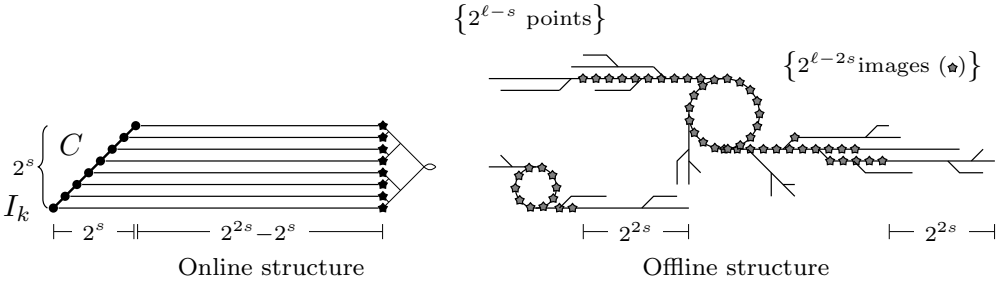
Directly matching the 2^s states computed by the MAC oracle on the challenge message with some states evaluated offline is too expensive. Therefore, we first compute and match the images of the states under iterations of a fixed function (as images match with higher probability than arbitrary states). Then, we use the initial matching of the images to efficiently match and recover an actual state that is computed by the MAC oracle on the challenge message. The 2-phase matching used in the attack resembles the approach used in [11,24], but the main difference is that the first phase of the matching does not exploit cycles in the functional graph. Moreover, the second phase of the matching is performed using an efficient binary search matching algorithm (Algorithm 1, whose pseudo-code is give in Appendix C). This algorithm can be simplified using a heuristic assumption on the number of nodes that need to be matched in the second phase (such an assumption was made explicitly in [11], and more implicitly in [6]). However, the analysis of Algorithm 1 is more rigorous and does not require any conjecture.

We denote the challenge message by C, and the first κ blocks of C by $C_{|\kappa}$. The details of the (first phase of the) universal forgery attack are as follows.

Attack 7: Universal forgery attack using chains (first phase)
Complexity $\tilde{O}(2^{\ell-s})$, with $s \leq \ell/7$ (min: $2^{6\ell/7}$)
Query length: $O(2^{2s})$

1. (online) Query the oracle with 2^s messages $M_i = C_{|i} \,\|\, [0]^{2^{2s}-i}$. Denote the set of (unknown) final states of the chains by Y. Build an online diamond filter for all states in Y.
2. (offline) Compute a structure of chains containing a total of $2^{\ell-s}$ points. Each chain is extended until it cycles or collides with a previous chain. Consider the set X of the 2^{2s}-iterates of f (namely images of $f^{2^{2s}}$ in the structure). According to Lemma 2, this set contains (no more than) about $2^{\ell-2s}$ distinct points. Build an online diamond filter for X.
3. (offline) Match all the points $x \in X$ with the 2^s points in Y.
4. (offline) For each match between $x \in X$ and an online state in Y (obtained using M_i), use an additional matching algorithm to test the actual message $C_{|i}$: call the binary search matching algorithm (Algorithm 1) with:
   - input message $M_i = C_{|i} \,\|\, [0]^{2^{2s}-i}$;
   - the tree rooted at x (obtained by disconnecting the edge between x and $f(x)$ from the graph and considering all the points that merge into x);
   - and distance $2^{2s} - i$.

   If the algorithm returns a match y', then with high probability the state obtained after processing $C_{|i}$ is equal to y'.



Online structure Offline structure

We efficiently detect a match between the challenge points (•) and the offline structure, by first matching X (⋆) and Y (⋆).

Algorithm 1: Binary search matching

Inputs: distance d, message M of length $d' > d$ whose last d blocks are zero, tree of chains computed with the zero block that merge into the root x

Output: node y in the tree with distance d to x that is equal to the state reached after evaluating $M_{|d'-d}$ (NULL if y does not exist)

1. Denote by $\textsc{Size}(u)$ the number of nodes in the tree whose root is u. Traverse the tree rooted at x (backwards) for at most d steps until a leaf or a collision is encountered:

- If a total of d steps were traversed, denote the node at distance d from x by z. Build a collision filter for z and test it on $M_{|d'-d}$. If they match, return $y = z$, otherwise go to Step 2.
- If a leaf is encountered (before d steps were traversed), denote the current node by z and go to Step 2.
- If colliding states z_1, z_2 such that $f(z_1) = f(z_2) = z$ are encountered, then if $\textsc{Size}(z_1) \geq \textsc{Size}(x)/2$ continue traversing z_1 in Step 1. Otherwise, if $\textsc{Size}(z_2) \geq \textsc{Size}(x)/2$ continue traversing z_2 in Step 1.[a]
  Otherwise (sizes of trees rooted at z_1 and z_2 are small), let $d(z)$ denote the distance of z from x. Build collision filters for z_1 and z_2 and test them on $M_{|d'-d(z_1)}$ (note that $d(z_1) = d(z_2) = d(z) + 1$). If there is no match, go to Step 2. Otherwise, assume without loss of generality that the match is with z_1. Call the algorithm recursively with input message $M_{|d'-d(z_1)}$, tree rooted at z_1, and distance $d' - d(z_1)$ (note that $\textsc{Size}(z_1) < \textsc{Size}(x)/2$).

2. Backtrack (by cutting off half of the tree): Denote by z the current node in the traversal. Let z' be the first ancestor of z (in the direction of the root x) for which there is a collision $f(z') = f(z'')$ for some z''. If there is no such z', return NULL. Otherwise, call the algorithm recursively with input message M, tree rooted at x with z' cut off, and distance d (note that $\textsc{Size}(z') \geq \textsc{Size}(x)/2$).

[a] The algorithm can be easily generalized to deal with collisions between $t \geq 2$ states by analyzing all t nodes in the t-way collision (looking for a child whose subtree size is at least half of the total tree size, and building filters for all the children if required). As the maximal value of t in an ℓ-bit random mapping is not expected to exceed ℓ, such collisions only add (up to) a logarithmic factor to the total complexity of the algorithm.

Analysis. The offline structure of Step 2 contains $2^{\ell-s}$ distinct points, and thus according to the birthday paradox, it covers one of the 2^s points of the challenge with high probability. In this case, the attack will successfully recover the state of the covered point with high probability, since the 2^{2s}-iterate of the covered point is also covered by the offline structure and will be matched in Step 3.[5]

The analysis of Algorithm 1 used in Step 4 is based on the following property: the size of the tree parameter is cut by at least half in every recursive call, where in each such recursive call we traverse at most $d \leq 2^{2s}$ nodes and compute and test a small constant number of collision filters. Since the initial tree has at

[5] The only case in which the 2^{2s}-iterate (denoted by x) of a covered point y is not covered by the offline structure, is when this iterate goes through the disconnected edge from x to $f(x)$ (namely, $f(x)$ is on a path from y to x whose length is at most 2^{2s}). This can only occur if x is on a cycle with less than 2^{2s} elements. Such (small) cycles are expected to belong to (small) connected components whose total size is $O(2^{2 \cdot 2s}) = O(2^{4s})$ (see [9]). On the other hand, we evaluate a total of $2^{\ell-s}$ nodes offline, and since $2^{4s} \ll 2^{\ell-s}$ for $s \leq \ell/7$, it is unexpected that the covered point y will belong to such a small component.

most 2^ℓ nodes, there are at most ℓ recursive calls in Algorithm 1 (for a single call in Step 4). The complexity of each recursive call is dominated by building a few offline collision filters, and therefore the complexity of Algorithm 1 is about $\ell \cdot 2^{\ell/2}$ (note that testing the filter online has complexity $2^{2s} < 2^{\ell/2}$, and traversing nodes in each recursive call has complexity at most $d \leq 2^{2s} < 2^{\ell/2}$).

Since we have 2^s points in Y and all the points in X are distinct, there are at most 2^s matches in Step 3, and the time complexity of Step 4 is at most $\ell \cdot 2^{\ell/2} \cdot 2^s = \ell \cdot 2^{\ell/2+s}$.

Complexity.

Step 1:	$2^{2s+s/2+\ell/2} = 2^{\ell/2+5s/2}$	**Step 2:**	$2^{\ell-s}$
Step 3:	$s \cdot 2^{\ell-2s+s} = s \cdot 2^{\ell-s}$	**Step 4:**	$\ell \cdot 2^{\ell/2+s}$

With $s \leq \ell/7$, we have $\ell/2 + 5s/2 \leq 6\ell/7 \leq \ell - s$; the complexity of the first phase of the universal forgery attack is $\tilde{O}(2^{\ell-s})$. Since the second phase has a similar complexity, this is also the complexity of the full attack.

9.2 A universal forgery attack using collisions

In this attack, we devise a different algorithm which recovers one of the states computed during the MAC computation on the challenge message. The main idea here is to begin with a state-recovery attack in order to determine the value of some online states. Then, we use iterates of a fixed function on the challenge states and on the known points, and detect (online) collisions between the two sets of iterates. This allows to determine the values of several iterates of challenge states, and removes the need for online filters (that are required in the first matching phase of the previous attack). Next, we evaluate some states offline and execute the second matching phase on offline iterates that match the online challenge iterates. This allow to recover an actual challenge state (the second phase matching uses Algorithm 1 as the previous attack).

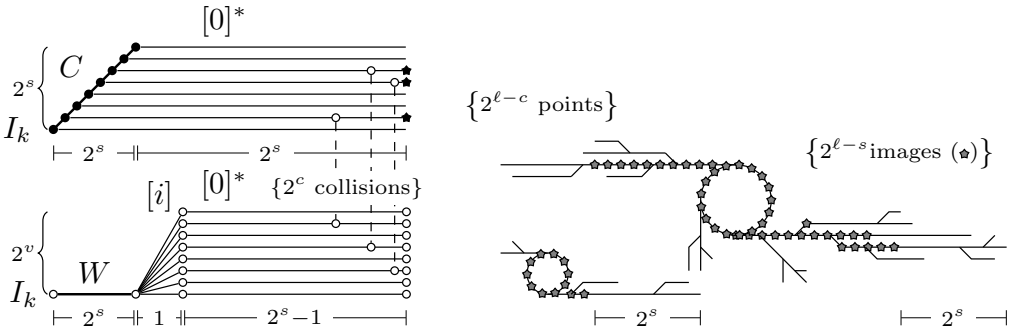
Attack 8: Universal forgery attack using collisions (first phase)
Complexity $O(2^{\ell-s/2})$, with $s \leq 2\ell/5$ (min: $2^{4\ell/5}$)
Query length: $O(2^s)$

1. (online) Query the oracle with 2^s messages $M_i = C_{|i} \parallel [0]^{2^{s+1}-i}$, and sort the tags.
2. (online) Execute state-recovery Attack 2 using messages of length $\min(2^s, 2^{\ell/3})$, and denote by W a message of length 2^s whose last computed state is recovered.[a]
3. (online) Query the oracle with 2^v messages $W_j = W \parallel [j] \parallel 0^{2^s-1}$, sort the tags, and locate 2^c collisions with the tags computed using the messages M_i. For each collision of tags between M_i and W_j, the

state reached after M_i is known, because the state reached after W is already known. We denote the set of those states as X.

4. (offline) Compute a structure of chains containing a total of $2^{\ell-c}$ points. Each chain is extended until it cycles or collides with a previous chain.
5. (online) For each offline point in the structure y which matches a point in X (corresponding to M_i), call the binary search matching algorithm (Algorithm 1) with inputs:
   - message M_i;
   - the tree rooted at y (obtained by disconnecting the edge between y and $f(y)$ from the graph and considering all the points that merge into y);
   - and distance $2^{s+1} - i$.

   If the algorithm returns a match y', then with high probability the state obtained after processing $C_{|i}$ is equal to y'.



We match the known points in X (★) and Y (★) in order to detect a match between the challenge points (●) and the offline structure.

[a] In case $s > \ell/3$, we first recover the last computed state of a message of size $2^{\ell/3}$, and then complement it arbitrarily to a length of 2^s.

Analysis. In Step 3 of the attack, we find 2^c collisions between pairs of chains, where the prefix of one chain in each pair is some challenge prefix $C_{|i}$. Thus, the 2^c collisions cover 2^c such challenge prefixes, and moreover, the offline structure computed in Step 4 contains $2^{\ell-c}$ points. Thus, according to the birthday paradox, with high probability the offline structure covers one of the states obtained after the computation of such a prefix $C_{|i}$. Since iterate $2^{s+1} - i$ of $C_{|i}$ is also covered by the offline structure,[6] then the state corresponding to $C_{|i}$ will be recovered in Step 5.

In order to calculate the value of c, note that the online structure computed in Step 1 contains 2^s chains, each of length at least 2^s, and thus another arbitrary chain of length 2^s collides with one of the chains in this structure at the same offset with probability of about $2^{2s-\ell}$ (see Lemma 1). Since the

[6] This occurs with high probability, and can be shown by analyzing the cycle structure of the graph as in the previous attack.

structure computed in Step 3 contains 2^v such chains, the expected number of detected collisions between the structures is $2^c = 2^{2s+v-\ell}$, *i.e.*, $c = 2s + v - \ell$.

In Step 5, we call Algorithm 1 at most 2^c times (at most once for each online collision), and each invocation has complexity $\ell \cdot 2^{\ell/2}$ (the analysis of Algorithm 1 is essentially the same as in the previous attack). In total, the complexity of Step 5 is $\ell \cdot 2^{c+\ell/2} = \ell \cdot 2^{2s+v-\ell/2}$.

Step 1:	2^{2s}	**Step 2:**	$\max(2^{\ell-s}, 2^{2\ell/3})$
Step 3:	2^{v+s}	**Step 4:**	$2^{\ell-c} = 2^{2\ell-2s-v}$
Step 5:	$\ell \cdot 2^{2s+v-\ell/2}$		

We balance steps 3 and 4 by setting $v + s = 2\ell - 2s - v$, or $v = \ell - 3s/2$. This yield $c = s/2$, a complexity of $2^{\ell-s/2}$ for steps 3 and 4, and $2^{\ell/2+s/2}$ for Step 5. This gives a total complexity of $O(2^{\ell-s/2})$ for any $s \leq 2\ell/5$.

10 Conclusions and Open Problems

In this paper, we provided improved analysis of `HMAC` and similar hash-based MAC constructions. More specifically, we devised the first state-recovery attacks for `HMAC` with hash functions based on the HAIFA mode, and provided improved tradeoffs between the message length and the complexity of state-recovery attacks for `HMAC` with Merkle-Damgård hash functions. Finally, we presented the first universal forgery attacks which can be applied with short queries to the MAC oracle. Since it is widely deployed, our attacks have many applications to `HMAC` constructions used in practice, built using `GOST`, the `SHA` family, and other concrete hash functions.

We give a more concrete application of some of these results in Section 8, with a key-recovery attack against `HMAC` with the new Russian standard `GOST` R 34.11-2012. In particular this attack requires a key-recovery attack against `HMAC` with HAIFA hash functions, and improved tradeoffs for short messages.

Our results raise several interesting future work items such as devising efficient universal forgery attacks on `HMAC` built using hash functions based on the HAIFA mode, or proving that this mode provides resistance against such attacks (perhaps under certain natural assumptions). At the same time, it would be useful to find additional applications of our algorithms to related cryptosystems (such as various modes of operation) where some level of security beyond the birthday bound is required.

References

1. Aumasson, J.P., Henzen, L., Meier, W., Phan, R.C.W.: SHA-3 proposal BLAKE. Submission to NIST (2008/2010), `http://131002.net/blake/blake.pdf`

2. Aumasson, J.P., Neves, S., Wilcox-O'Hearn, Z., Winnerlein, C.: BLAKE2: Simpler, Smaller, Fast as MD5. In: Jr., M.J.J., Locasto, M.E., Mohassel, P., Safavi-Naini, R. (eds.) ACNS. Lecture Notes in Computer Science, vol. 7954, pp. 119–135. Springer (2013)
3. Bellare, M.: New Proofs for. In: Dwork, C. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 4117, pp. 602–619. Springer (2006)
4. Bellare, M., Canetti, R., Krawczyk, H.: Keying Hash Functions for Message Authentication. In: Koblitz, N. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 1109, pp. 1–15. Springer (1996)
5. Biham, E., Dunkelman, O.: A Framework for Iterative Hash Functions - HAIFA. IACR Cryptology ePrint Archive, Report 2007/278 (2007)
6. Dinur, I., Leurent, G.: Improved Generic Attacks against Hash-Based MACs and HAIFA. In: Garay and Gennaro [10], pp. 149–168
7. Dolmatov, V., Degtyarev, A.: GOST R 34.11-2012: Hash Function. RFC 6986 (Informational) (Aug 2013), http://www.ietf.org/rfc/rfc6986.txt
8. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The Skein hash function family. Submission to NIST (2008/2010), http://skein-hash.info
9. Flajolet, P., Odlyzko, A.M.: Random Mapping Statistics. In: Quisquater, J., Vandewalle, J. (eds.) Advances in Cryptology - EUROCRYPT '89, Workshop on the Theory and Application of of Cryptographic Techniques, Houthalen, Belgium, April 10-13, 1989, Proceedings. Lecture Notes in Computer Science, vol. 434, pp. 329–354. Springer (1989)
10. Garay, J.A., Gennaro, R. (eds.): Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I, Lecture Notes in Computer Science, vol. 8616. Springer (2014)
11. Guo, J., Peyrin, T., Sasaki, Y., Wang, L.: Updates on Generic Attacks against HMAC and NMAC. In: Garay and Gennaro [10], pp. 131–148
12. Guo, J., Sasaki, Y., Wang, L., Wang, M., Wen, L.: Equivalent Key Recovery Attacks Against HMAC and NMAC with Whirlpool Reduced to 7 Rounds. In: Cid, C., Rechberger, C. (eds.) Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers. Lecture Notes in Computer Science, vol. 8540, pp. 571–590. Springer (2014)
13. Guo, J., Sasaki, Y., Wang, L., Wu, S.: Cryptanalysis of HMAC/NMAC-Whirlpool. In: Sako and Sarkar [25], pp. 21–40
14. Joux, A.: Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In: Franklin, M.K. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 3152, pp. 306–316. Springer (2004)
15. Kelsey, J., Kohno, T.: Herding Hash Functions and the Nostradamus Attack. In: Vaudenay, S. (ed.) EUROCRYPT. Lecture Notes in Computer Science, vol. 4004, pp. 183–200. Springer (2006)
16. Kelsey, J., Schneier, B.: Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work. In: Cramer, R. (ed.) EUROCRYPT. Lecture Notes in Computer Science, vol. 3494, pp. 474–490. Springer (2005)
17. Kortelainen, T., Kortelainen, J.: On Diamond Structures and Trojan Message Attacks. In: Sako and Sarkar [25], pp. 524–539
18. Leurent, G., Peyrin, T., Wang, L.: New Generic Attacks Against Hash-based MACs. IACR Cryptology ePrint Archive 2014, 406 (2014), http://eprint.iacr.org/2014/406
19. Leurent, G., Peyrin, T., Wang, L.: New Generic Attacks against Hash-Based MACs. In: Sako and Sarkar [25], pp. 1–20
20. Love, E.R.: Some Logarithm Inequalities. The Mathematical Gazette 64(427), 55–57 (1980), http://www.jstor.org/stable/3615890
21. Naito, Y., Sasaki, Y., Wang, L., Yasuda, K.: Generic State-Recovery and Forgery Attacks on ChopMD-MAC and on NMAC/HMAC. In: Sakiyama, K., Terada, M. (eds.) IWSEC. Lecture Notes in Computer Science, vol. 8231, pp. 83–98. Springer (2013)
22. van Oorschot, P.C., Wiener, M.J.: Parallel Collision Search with Cryptanalytic Applications. J. Cryptology 12(1), 1–28 (1999)
23. Peyrin, T., Sasaki, Y., Wang, L.: Generic Related-Key Attacks for HMAC. In: Wang, X., Sako, K. (eds.) ASIACRYPT. Lecture Notes in Computer Science, vol. 7658, pp. 580–597. Springer (2012)

24. Peyrin, T., Wang, L.: Generic Universal Forgery Attack on Iterative Hash-Based MACs. In: Nguyen, P.Q., Oswald, E. (eds.) Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8441, pp. 147–164. Springer (2014)
25. Sako, K., Sarkar, P. (eds.): Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II, Lecture Notes in Computer Science, vol. 8270. Springer (2013)
26. Tsudik, G.: Message authentication with one-way hash functions. SIGCOMM Comput. Commun. Rev. 22(5), 29–38 (Oct 1992)
27. Yasuda, K.: "Sandwich" Is Indeed Secure: How to Authenticate a Message with Just One Hashing. In: Pieprzyk, J., Ghodosi, H., Dawson, E. (eds.) ACISP. Lecture Notes in Computer Science, vol. 4586, pp. 355–369. Springer (2007)

A The Second Phase of the Universal Forgery Attacks [24]

In this section we describe the second phase of our universal forgery attacks, which is identical to the second phase of the attack of [24]. Recall the we are given a challenge message $C = m_1 \parallel m_2 \parallel \ldots \parallel m_{2^s}$ of length 2^s blocks and our goal is to predict the tag of C by querying the MAC oracle with queries that are different from C. We denote the internal states computed by the MAC oracle on the challenge C by $y_0, y_1, \ldots, y_{2^s}$. In the first phase of the attack, we recover one of these internal states, which we denote by y_p. We assume that $p \leq 2^{s-1}$. This can be assured by running the first phase of the attack only on the first half of the blocks of C, which results in a small constant penalty in the complexity of the attack.

In the second phase of the attack we use a second-preimage attack on long messages in order to generate a different message C' which has the same tag as the challenge. The second-preimage attack is based on the one by Kelsey and Schneier [16]. The attack uses a structure known as an (i, j)-expandable message (for integers $i < j$), proposed in [16]. An (i, j)-expandable message is composed of an arbitrary initial state x and a final state z where for each integer d such that $i \leq d \leq j$, there is an efficiently computable message of d blocks $M_d = m'_1 \parallel m'_2 \parallel \ldots \parallel m'_d$ that links x to z, namely, $z = h(\ldots h(h(x, m'_1), m'_2) \ldots, m'_d)$. Assuming that $i \leq l/2$ (which is the case in our attack), an $(i, 2^i)$-expandable message can be constructed from any initial state x in complexity of $\ell \cdot 2^{\ell/2}$. The details of how to construct this structure are found in [16].

Algorithm 2: Universal Forgery Attack (second phase)
Complexity $\tilde{O}(2^{\ell-s})$, with $s \leq \ell/2$

1. (offline) Compute the states $y_p, y_{p+1}, \ldots, y_{p+2^{s-1}}$ and store them in a sorted table.
2. (offline) Build an $(s-1, 2^{s-1})$-expandable message starting from y_p and denote its final state by z.
3. (offline) For about $2^{\ell-s}$ values of the single-block $[i]$, compute $h(z, [i])$ and search the sorted table for y_j such that $y_j = h(z, [i])$ and $j \geq p + s$. Once such i and j are found, continue to the next step.
4. (online) Let M_{j-1-p} be the message of $j-1-p$ blocks that links y_p to z, computed from the expandable message. Let $C' = C_p \parallel M_{j-p-1} \parallel [i] \parallel m_{j+1} \parallel m_{j+2} \parallel \ldots \parallel m_{2^s}$ be the 2^s-block message computed by concatenating the p-block prefix of C with $M_{j-p-1} \parallel [i]$ and the suffix of $2^s - j$ blocks of C. Query the MAC oracle with C' and denote the result by T.
5. (offline) Output T as the tag for C.

Complexity.

Step 1:	2^{s-1}	**Step 2:**	$\ell \cdot 2^{\ell/2}$
Step 3:	$2^{\ell-s}$	**Step 4:**	2^s
Step 5:	2^s		

For $s \leq \ell/2$ the total complexity is $\tilde{O}(2^{\ell-s})$ as required.

B Concrete Complexity of HMAC-Streebog Key-Recovery

In this section we calculate with better precision (taking into account constants and logarithmic factors) an upper bound on the expected complexity of Attack 6 (applied to Streebog with $\ell = 512$). While the analysis can be refined to give slightly better complexity, our calculations demonstrate that the constants involved in our attacks are indeed small.

Attack 6 is dominated by Step 0, which uses Attack 5, with $s = \ell/10$. Therefore, we start with a concrete analysis of Attack 5.

Attack 5. The most expensive steps are 2 and 4, where we match two sets by building and using a diamond filter. In order to optimize the attack, we use parameters $c_1 = \ell/2 - s$ and $c_2 = \ell/2 + \alpha$ for a small constant α (*e.g.* $\alpha = 4$), and run steps 1 and 3 until we have the required number of collisions (2^{c_1} and 2^{c_2}).

The expected value of u required to generate 2^{c_1} collisions in Step 1 is roughly $u = (c_1 + \ell - s + 1)/2$, accounting for the factor $\frac{1}{2}$ in the number of message pairs. Similarly, the expected value of t in Step 3 is $t = (c_2 + \ell - s + 1)/2$. This results in a complexity that is still negligible compared to steps 2 and 4, in spite of the small constants added ($2^{3\ell/4+1/2}$ for Step 1 and $2^{3\ell/4+(s+\alpha+1)/2}$ for Step 3).

With these parameters the online diamond filter for the set of size 2^{c_1} will have depth $\lceil c_1 \rceil$, and require less than $8\frac{e}{e-1}2^{c_1/1+\ell/2}$ evaluations of the MAC (following [17]). Since each query has 2^s blocks, the total complexity is $8\frac{e}{e-1}2^{s+c_1/2+\ell/2} < 2^{3\ell/4+s/2+3.7}$ for Step 2.

Finally, in Step 4, we use only collisions with an offset d such that $d \geq 2^{s-1}$, to simplify the analysis (we keep about $2^{c_2-1} = 2^{\ell/2+3}$ offline collisions). Following Lemma 4, we fix an offline collision $\hat{x}$ (with preimages x_d and x'_d) and we evaluate the probability that a random pair of chains y_0, y'_0 reaches the same collision, under the condition that y_0, y'_0 collide at the same offset d. Let A denotes the event that y_0, y'_0 collide on $\hat{x}$ and B the event that they collide at offset d on an arbitrary point (all collisions are same-offset collisions). We first evaluate the probability that the chains starting at y_0 and y'_0 reach x_d or x'_d, and then the probability that y_0, y'_0 collide on $\hat{x}$ (namely $\Pr[A]$).

$$\Pr[y_0 \not\rightsquigarrow x_d] \leq (1 - 2^{-\ell})^{2^{s-1}} \leq e^{-2^{s-1-\ell}}$$

$$\Pr[y_0 \rightsquigarrow x_d] = 1 - \Pr[y_0 \not\rightsquigarrow x_d] \geq 1 - e^{-2^{s-1-\ell}} \approx 2^{s-1-\ell}$$

$$\Pr[A] \geq \Pr[y_0 \rightsquigarrow x_d] \cdot \Pr[y'_0 \rightsquigarrow x'_d] + \Pr[y'_0 \rightsquigarrow x_d] \cdot \Pr[y_0 \rightsquigarrow x'_d] \geq 2^{2s-2\ell-1}$$

We have $\Pr[B] \leq 2^{-\ell}$, hence $\Pr[A|B] = \Pr[A \cap B]/\Pr[B] \geq 2^{2s-\ell-1}$. Therefore, the matching of Step 4 is expected to succeed after less than $2^{\ell-2s+1}$ attempts on average (the actual complexity without the restriction $d \geq 2^{s-1}$ is slightly smaller). Since each match requires to follow a path in a diamond of length c_1, the total complexity is less than $c_1 \cdot 2^{\ell-2s+1} \leq \ell \cdot 2^{\ell-2s}$ on average. This results in the following complexities:

Step 1:	$2^{3\ell/4+1/2}$	**Step 2:**	$2^{3\ell/4+s/2+3.7}$
Step 3:	$2^{3\ell/4+s/2+(\alpha+1)/2}$	**Step 4:**	$\ell \cdot 2^{\ell-2s}$

In particular, with $\ell = 512$ and $s = 54$, we obtain a complexity of $2^{415.1}$.

Attack 6. Let us now examine the full key-recovery attack, using a short message M_1 of length 2^s. We run Step 2 until we gather $2^{2\ell/5}$ collisions, which requires about $2^{7\ell/10+1/2}$ calls to the MAC on average. Similarly, we run Step 3 until we recover the key; this should require $2^{3\ell/5}$ collisions on average, *i.e.* $2^{4\ell/5+1/2}$ evaluations of the compression function.

Since Step 0 still dominates, we use $s = 54$ to minimize the complexity:

Step 0:	$2^{415.1}$	**Step 1:**	$\ell \cdot 2^{\ell/2} = 2^{247}$
Step 2:	$2^{s+7\ell/10+1/2} = 2^{412.9}$	**Step 3:**	$2^{4\ell/5+1/2} = 2^{410.1}$

Therefore, the expected complexity is less than 2^{416}.

Finally, recall from Section 2 that our attacks use a modified scheme $\mathrm{MAC}_2(K, M) = \mathrm{MAC}_1(K, M \parallel [0]) \parallel \mathrm{MAC}_1(K, M \parallel [1])$, which results in a multiplicative penalty of at most 2. Hence, the total expected complexity is less than 2^{417}.

C Pseudo-code

Algorithm 1 Binary search matching

```
function SEARCH(x, d)                                   ▷ The message is a global parameter
    if |NEXT(x) = 0| then
        return NULL
    end if
    z ← x
    i ← d
    while i > 0 do
        if |NEXT(z) = 0| then
            return CUT(z)
        else if |NEXT(z) = 1| then
            z ← NEXT(z)
        else if |NEXT(z) = 2| then
            z_1, z_2 ← NEXT(z)
            if SIZE(z_1) > SIZE(x)/2 then
                z ← z_1
            else if SIZE(z_2) > SIZE(x)/2 then
                z ← z_2
            else
                if TEST (z_1) then                      ▷ Compare z_1 to a message node using a filter
                    return SEARCH(z_1, i)               ▷ We have SIZE(z_1) ≤ SIZE(x)/2
                else if TEST (z_2) then                 ▷ Compare z_2 to a message node using a filter
                    return SEARCH(z_2, i)               ▷ We have SIZE(z_2) ≤ SIZE(x)/2
                else
                    return CUT(z)
                end if
            end if
        else if |NEXT(z) = 3| then
            ...                                         ▷ Handling a t > 2-collision is similar
        end if
        i ← i − 1
    end while
    if TEST (z) then
        return z
    else
        return CUT(z)
    end if
end function

function CUT(z)
    while |NEXT(PREV(z))| = 1 do
        z ← NEXT(PREV(z))
    end while
    Remove z subtree                                    ▷ We have SIZE(z) ≥ SIZE(x)/2
    return SEARCH(x, d)
end function
```

On the Practical (In-)Security of 64-bit Block Ciphers

Collision Attacks on HTTP over TLS and OpenVPN

Karthikeyan Bhargavan
Inria, France
karthikeyan.bhargavan@inria.fr

Gaëtan Leurent
Inria, France
gaetan.leurent@inria.fr

ABSTRACT

While modern block ciphers, such as AES, have a block size of at least 128 bits, there are many 64-bit block ciphers, such as 3DES and Blowfish, that are still widely supported in Internet security protocols such as TLS, SSH, and IPsec. When used in CBC mode, these ciphers are known to be susceptible to collision attacks when they are used to encrypt around 2^{32} blocks of data (the so-called birthday bound). This threat has traditionally been dismissed as impractical since it requires some prior knowledge of the plaintext and even then, it only leaks a few secret bits per gigabyte. Indeed, practical collision attacks have never been demonstrated against any mainstream security protocol, leading to the continued use of 64-bit ciphers on the Internet.

In this work, we demonstrate two concrete attacks that exploit collisions on short block ciphers. First, we present an attack on the use of 3DES in HTTPS that can be used to recover a secret session cookie. Second, we show how a similar attack on Blowfish can be used to recover HTTP BasicAuth credentials sent over OpenVPN connections. In our proof-of-concept demos, the attacker needs to capture about 785GB of data, which takes between 19-38 hours in our setting. This complexity is comparable to the recent RC4 attacks on TLS: the only fully implemented attack takes 75 hours. We evaluate the impact of our attacks by measuring the use of 64-bit block ciphers in real-world protocols. We discuss mitigations, such as disabling all 64-bit block ciphers, and report on the response of various software vendors to our responsible disclosure of these attacks.

1. INTRODUCTION

Internet protocols such as TLS [13], SSH [34], and IPsec [18] are *agile*, in the sense that they are designed to support a wide variety of *ciphersuites*: combinations of key exchange protocols, encryption schemes, authentication modes, etc. Each protocol implementation may choose to support a different subset of ciphersuites, but two implementations can still interoperate if they can negotiate a common ciphersuite. When it works well, protocol agility can enable a graceful transition from old cryptographic algorithms to new ones. A server can, for example, offer AES-GCM to modern clients while still supporting legacy ciphers like 3DES for older clients that have not yet been upgraded. However, a negative consequence of agility is that old ciphers may never be removed, resulting in implementations that support a few strong modern ciphers, followed by a long tail of obsolete ciphers that are still supported for backwards compatibility, but are known to be cryptographically weak. For example, the OpenSSL library supports five versions of TLS and hundreds of ciphersuites, even though many of these ciphersuites include weak algorithms like RC4 and MD5.

There are several reasons why practitioners do not consider theoretical weaknesses in cryptographic to be sufficient for their removal from mainstream protocols. First, even if an obsolete primitive is supported, it will typically only be negotiated if one of the parties does not support a modern alternative, in which case, the obsolete cipher is still better than nothing. Second, the attack may not be applicable to the way the primitive is used in the protocol, or it may require too much computation to be considered practical. Third, the attack may require special knowledge about the structure or the content of the plaintext stream which may be difficult to obtain in general. Consequently, protocol implementations tend to support legacy ciphers until a concrete attack is demonstrated in a common usage of the protocol.

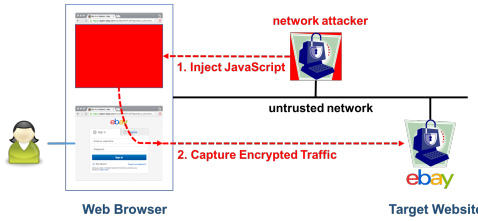


Figure 1: The Beastly attacker model for HTTPS: A network attacker injects JavaScript on some window in the user's browser, which then makes cross-origin HTTPS requests to a target website. The browser attaches a secret cookie or HTTP BasicAuth password to authenticate each request. The attacker observes the encrypted traffic stream and tries to recover the secret.

CCS '16, October 24–28, 2016, Vienna, Austria.

ACM ISBN 978-1-4503-4139-4/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2976749.2978423

Attacks on Legacy Crypto. A series of recent attacks on TLS have conclusively shown that this line of defense for legacy ciphers is fatally flawed. Leaving old protocol versions and obsolete cryptographic algorithms enabled can lead to devastating attacks, such as BEAST [15] (on CBC encryption in TLS 1.0), CRIME [28] (on TLS compression), Lucky13 [4] (on CBC padding in TLS 1.2), POODLE [11] (on CBC padding in SSL 3.0), RC4 NOMORE [3, 19, 32] (on keystream biases in RC4), FREAK [9] (on export-grade RSA keys), Logjam [2] (on export-grade Diffie-Hellman groups), SLOTH [10] (on MD5 signatures), and DROWN [5] (on RSA-PKCS#1v1.5 encryption in SSL 2.0).

These attacks follow a common pattern. Each attack is based on a classic cryptographic weakness that was known to cryptographers for years or even decades in advance, but had not been demonstrated in real-world TLS scenarios. So these works showed that with modern network speeds and computing power, the computational effort required to exploit the theoretical weakness is feasible even for academic researchers. Furthermore, many of these attacks (e.g. BEAST, CRIME, RC4 NOMORE) can be made more efficient by relying on a new HTTPS attacker model introduced by BEAST. Figure 1 depicts this Beastly attacker model, where a network attacker uses JavaScript to implement an adaptive chosen-plaintext adversary who can mix in his plaintext with the user's secret values, and then use his network vantage point to actively or passively attack the ciphertext.

The final element of each attack is finding clients and servers who are willing to negotiate the legacy cipher. Some of the attacked algorithms (e.g. RC4) and protocol versions (e.g. SSL 3.0) were highly popular before the attack and have since been deprecated. Other weak algorithms (e.g. export ciphers) and versions (e.g. SSL 2.0) had already been deprecated years ago and were never expected to be negotiated between modern clients and servers. Surprisingly however, in many cases the attacker is able to force the negotiation of a weak algorithm via a *downgrade attack* by exploiting protocol flaws (e.g. Logjam, SLOTH) or implementation bugs (e.g. FREAK, DROWN). One lesson to take from these downgrade attacks is that relying on the negotiation protocol to avoid legacy ciphers is error-prone and dangerous. When a concrete attack against a legacy cipher is discovered, the only safe mitigation is to fully remove the weak cipher from all implementations.

These previous attacks evaluated a variety of obsolete cryptographic constructions and provided enough evidence of their dangers for them to be disabled in popular implementations. We believe that it is important for the research community to be vigilant and to continue to investigate the strength of the weakest ciphers still supported by mainstream protocols. For example, the attacks on RC4 have resulted in RC4 being removed as the default fallback cipher for TLS implementations, and this role has now fallen to 3DES, which is a 28 year old 64-bit block cipher with known weaknesses. However, the practical security of 64-bit block ciphers as used in TLS and other protocols has not been previously studied. This work aims to address this gap and to provide concrete attacks and firm guidance on the use of such ciphers.

Collision Attacks on 64-bit Block Ciphers. The security of a block cipher is often reduced to the key size k: the best attack should be the exhaustive search of the key, with complexity 2^k. However, the block size n is also an important security parameter, defining the amount of data that can be encrypted under the same key. This is particularly important when using common modes of operation: we require block ciphers to be secure with up to 2^n queries, but most modes of operation (*e.g.* CBC, CTR, GCM, OCB, etc.) are unsafe with more than $2^{n/2}$ blocks of message (the so-called birthday bound). In particular, when these many blocks are encrypted under a key in CBC mode, the probability of collisions between two ciphertext blocks becomes significant, and each ciphertext collision reveals the XOR of the two corresponding plaintext blocks from the stream. Consequently, if the attacker can guess one of the plaintext blocks, he can recover the plaintext in the other block. While this attack is well-known to researchers [30, 25], it is often disregarded by practitioners because it require known plaintext, and reveals only a few bits of a large datastream.

With a modern block cipher with 128-bit blocks such as AES, the birthday bound corresponds to 256 EB (2^{68} bytes). While this is leaves a safe margin for common uses of cryptography today, the resulting security is still lower than could be expected of 128-bit security and might be an issue in the future. As a point of comparison, Google is estimated to store around 16 EB (2^{64} bytes), and the global Internet traffic is expected to reach 1 ZB (2^{70} bytes) per year in 2016.

More strikingly, the birthday bound corresponds to only 32 GB for a block cipher with 64-bit blocks, and there are many real-world protocol scenarios in which this amount of data may be sent under a 64-bit cipher: e.g. 3G telephone connections (UMTS), which are encrypted with KASUMI, OpenVPN tunnels, which uses Blowfish as the default cipher, and Internet standards like TLS, SSH, and IPsec that use 3DES and Blowfish as legacy ciphers for interoperability. In all these scenarios, 32 GB of data can be transferred in less than one hour with a fast connection.

More recently, 64-bit ciphers have become common in the field of *lightweight cryptography* for applications, such as embedded systems and smart devices, where resources are too constrained to use conventional block ciphers. For instance, PRESENT [12] and HIGHT [20] have been standardized by ISO recently (in ISO/IEC 29192 and ISO/IEC 18033-3, respectively).

Our Results. In this work, we study the concrete impact of birthday attacks against 64-bit block-ciphers as used in popular protocols. We present the first concrete attacks based on ciphertext collisions on mainstream Internet protocols.

We focus on two concrete use cases. First, we consider the use of OpenVPN to secure HTTP connections between a web browser and a website, where the website asks the user for HTTP BasicAuth (user/password) authentication. OpenVPN uses Blowfish in CBC mode by default, and we show how one can use the Beastly attacker scenario to generate a lot of traffic including the BasicAuth in every request, until collisions collisions in the Blowfish ciphertext reveal the BasicAuth token (*i.e.* the password). We demonstrate a proof-of-concept attack between a Firefox client and an nginx server connected via OpenVPN. The attack requires about 785GB, which takes 19 hours in our setting to recover the 16-byte authorization token.

Second, we consider the use of TLS to secure HTTP connections between a browser and an HTTPS website that sets secure session cookies on the browser. We find that 3DES is supported by nearly 90% of HTTPS servers and by all mainstream TLS clients. In practice, however, 3DES is less preferred than AES and negotiated by only 1-2% of

HTTPS connections. For such connections, we demonstrate an attack, again using the Beastly attacker setup, that can recover the secret 16-byte session cookie sent by the browser with every HTTPS request to the target server. We demonstrate the attack between a Firefox browser and an IIS 6.0 server (Windows Server 2003 R2 SP2), both in their default up-to-date configuration. This attack also requires 785GB, which takes 38 hours in our setting.

Our attacks demonstrate that collision-based attacks on 64-bit block ciphers are practically exploitable. We suggest two mitigations for the attack. Considering the low negotiation probability of 3DES in TLS, we recommend disabling it in popular clients and servers. For protocols where 64-bit ciphers cannot yet be deprecated, we recommend using strict rekeying limits to mitigate the attack. In response to our attacks, mainstream TLS libraries, web browsers, and OpenVPN have begun implementing these mitigations.

Outline. Section 2 provides background on collision attacks on block ciphers and estimates their success probability under various conditions. Section 3 investigates the usage of 64-bit ciphers in various Internet protocols, and specifically in HTTPS. Section 4 shows how block cipher collisions in the encrypted stream can be used to attack HTTP. Section 4.4 and 4.5 describe our proof-of-concept attacks on OpenVPN and HTTPS, respectively. Section 5 discusses the impact of our attacks and their mitigations.

2. COLLISION ATTACKS ON BLOCK CIPHERS

Block ciphers like AES and 3DES are widely used for symmetric encryption in security protocols. Mathematically, a block cipher with a κ-bit key and n-bit blocks defines of family of permutations of n-bit strings, indexed by the key. The main security notion is that a block cipher should behave like a pseudo-random permutation (PRP) family: the block cipher E_k with a random key k should be hard to distinguish from a random permutation.

Block ciphers operate on blocks of fixed size, n bits, usually with $n = 64$ (DES, 3DES, Blowfish, IDEA, KASUMI, PRESENT, ...) or $n = 128$ (AES, Camellia, Twofish, Serpent, SEED, ...). In order to deal with arbitrary-length messages, they are used within a *mode* of operation. A mode divides the message M into n-bit blocks m_i, and process the blocks one by one through the block cipher E with various chaining rules, to produce ciphertext blocks c_i and/or an authentication tag τ. Several modes have been widely studied and standardized [16] for encryption (CBC, CFB, OFB, CTR), authentication (CBC-MAC, PMAC) and authenticated encryption (OCB, CCM, GCM).

2.1 Security at the Birthday Bound

While block ciphers are required to resist any attack with up to 2^n data and 2^κ time, most modes of operation are only proven secure up to $2^{n/2}$ blocks of plaintext, a limit that is commonly called the *birthday bound*. Indeed, there are attacks matching this limit. In CBC mode, the probability of collisions between two n-bit ciphertext blocks becomes significant after $2^{n/2}$ blocks because of the birthday paradox. Rogaway summarizes the situation as follows [30, page 36]:

> In general, unless special efforts are taken, almost any mode of operation based on an n-bit block-cipher will admit attacks that employ $\sigma = 2^{n/2}$ blocks. [...]
>
> Do birthday-bound attacks on CBC, CFB, and OFB actually matter? They are of relatively little concern when the blockcipher has a blocksize of $n = 128$ bits, but the attacks can be a serious concern when employing a blockcipher of $n = 64$ bits, requiring relatively frequent rekeying to keep $\sigma \ll 2^{32}$.

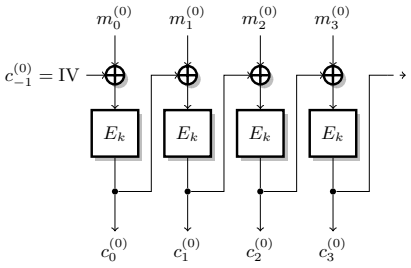


Figure 2: CBC mode

More recently, McGrew investigated plaintext recovery attacks on 64-bit ciphers in CBC, CFB, and CTR modes. He studied the mount of information leakage by assuming that the attacker can observe all the ciphertext in an encrypted stream and he has some (incomplete) knowledge of the plaintext [25]. This theoretical analysis resulted in new security recommendations for IPsec [26], but they did not trigger any significant changes in mainstream deployments of 64-bit ciphers in IPsec, TLS, or SSH. Arguably the main reason for the limited impact of these attacks is that they were never implemented against a popular protocol under real-world conditions.

Following McGrew, we explain theoretical attacks against the CBC and CTR modes (the two most widely used encryption modes) in more detail below. In the rest of the paper, we will show that these attacks can be exploited to mount concrete attacks on HTTP over TLS and OpenVPN.

2.2 Plaintext recovery attack against CBC

The CBC mode is one of the oldest encryption modes, and still widely used. The message M is divided into blocks $m_i, 0 \le i < \ell$ (see Figure 2) and is encrypted as:

$$c_i = E_k(m_i \oplus c_{i-1}),$$

where c_{-1} is an initialization value usually denoted as IV. For simplicity, here we assume that all message sizes are a multiple of the block size.

When encrypting a sequence of messages $M^{(j)}$, each message should be encrypted with a fresh random IV $c_{-1}^{(j)}$. It is important that the IV can not be predicted by an adversary who can encrypt chosen messages; otherwise, it can lead to attacks, as first observed by Rogaway in 1995 [29], and subsequently exploited in the BEAST attack [15] in 2011 on TLS versions prior to 1.1.

CBC has been proven secure up to $2^{n/2}$ blocks of messages [7, 27]. On the other hand there is a simple birthday attack against CBC: after $2^{n/2}$ message blocks encrypted

with the same key (in the same message or in different messages), a collision between two ciphertext blocks $c_i^{(j)} = c_{i'}^{(j')}$ is expected. Since E_k is a permutation, a collision in the output means that the inputs are the same ($m_i^{(j)} \oplus c_{i-1}^{(j)} = m_{i'}^{(j')} \oplus c_{i'-1}^{(j')}$), which reveals the xor of two plaintext blocks:

$$m_i^{(j)} \oplus m_{i'}^{(j')} = c_{i-1}^{(j)} \oplus c_{i'-1}^{(j')}.$$

Success probability. In general, with 2^d blocks of data, the probability that there is at least one collision is (using $D = 2^d$ and $N = 2^n$):

$$\begin{aligned} p &= 1 - \prod_{i=0}^{D-1} \frac{N-i}{N} \\ &\geq 1 - \prod_{i=0}^{D-1} e^{-i/N} && \text{using } 1 - x \leq e^{-x} \\ &\geq 1 - e^{-D(D-1)/2N} \end{aligned}$$

In particular, with $d > n/2$, we have $p \geq 1 - e^{-1/2} \approx 0.39$. With larger values of d, the probably grows very close to 1:

$$\begin{aligned} p &\geq 1 - \frac{2N}{D(D-1)+2N} && \text{using } 1 - e^{-x} > \frac{x}{x+1} \\ &\geq 1 - \frac{2N}{D^2} && \text{since } D^2 < D(D-1) + 2N \\ p &\geq 1 - 2^{n-2d+1} \end{aligned}$$

On the other hand, when $d < n/2$ we can use:

$$p \geq \frac{D(D-1)}{4N} \approx 2^{2d-n-2} \qquad \text{using } e^{-x} < 1 - x/2$$

Furthermore, with 2^d blocks of data the expected number of collisions is roughly 2^{2d-n-1}.

2.3 Distinguishing attack against CTR

The counter mode CTR is another popular mode of operation for encryption, first proposed by Diffie and Hellman [14]. In particular, it provides the encryption part of authenticated encryption modes CCM and GCM, and a variant of counter mode is used in 3G telephony (f8). The counter mode actually builds a stream cipher out of a block cipher: a counter is encrypted to generate key-stream that is xored with the message. More precisely, the message M is divided into blocks $m_i, 0 \leq i < \ell$ which are encrypted as:

$$c_i = E_k(\text{IV} + i) \oplus m_i,$$

where IV is the initial value of the counter (see Figure 3).

If the encryption function E_k is a pseudo-random *function*, rather than a pseudo-random *permutation*, then CTR is secure up to 2^n blocks [6]. However, in the common case where E_k is a permutation, CTR is proven secure only up to $2^{n/2}$ blocks, and there is matching distinguishing attack with complexity $2^{n/2}$. If a message where all the plaintext blocks are equal is encrypted, then all the ciphertext blocks will be pairwise different. Indeed, all the $E_k(\text{IV} + i)$ are unique, because E_k is a permutation. This enables the attacker to distinguish the counter mode from an ideal encryption scheme, because a collision between two ciphertext blocks becomes likely (with probability about 0.39). More interestingly, if the attacker has prior knowledge about the plaintext (such as a dictionary), he can use the distinguishing attack to eliminate *impossible values* to eventually guess the right plaintext [25].

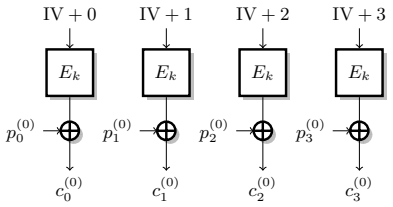


Figure 3: CTR mode

It is interesting to compare the collision attacks against CBC and CTR. Both attacks have the same complexity, and show that the proofs are tight. However, the loss of security is quite different: an attacker can immediately recover some bits of information from the attack against CBC, while the attack against the counter mode requires a lot more work to recover any plaintext. In the remainder of this paper, we investigate on the practical impact of the CBC attack, but similar techniques (with higher complexity and more assumptions on the plaintext) may be used to attack CTR and GCM mode encryption in protocols like TLS and SSH.

2.4 Rekeying before the Birthday Bound

The main countermeasure against birthday attacks is to change the key frequently, before reaching the birthday bound under any single key. For instance, if the key is changed after $2^{n/2-16}$ blocks, a birthday attack against a given only succeeds with probability 2^{-32}, and an attacker has to collect 2^{32} different sessions to mount an attack with high probability.

If implemented properly, this is an effective security measure to strengthen block ciphers with a small block-size. We stress than the key must be changed *well before* $2^{n/2}$ blocks, despite misleading recommendations in published standards. For instance, ISO recommends [21]:

> Generally, for a block cipher with block size of n bits, the maximum amount of plaintext that can be encrypted before rekeying must take place is $2^{(n/2)}$ blocks, due to the birthday paradox. As long as the implementation of a specific block cipher do not exceed these limits, using the block cipher will be safe.

This is *incorrect*, because collisions can be found with high probability (close to 0.39) with only $2^{n/2}$ blocks. Indeed, we will show that our concrete attacks also work against connections that are rekeyed close to the birthday bound.

2.5 Towards a Practical Attack on CBC

The CBC attack reveals the xor of two plaintext block, and in many contexts, this may not be sufficient for a practical attack. However, an effective attack can be mounted when the following conditions are fulfilled: (1) some high-value secret data is sent repeatedly; and (2) some fraction of the plaintext is known. In this case, there is a chance that a collision leaks the xor between a valuable secret and a known block; this would immediately reveal the secret. More precisely, this attack is expected to succeed (*i.e.* to recover one block of a high-value secret) with high probability as

soon as 2^s blocks of secret and 2^t known blocks are encrypted, with $s + t \geq n$.

To mount the attack in practice, an adversary would collect and store all the traffic, then sort the ciphertext blocks to find collisions. Assuming that the adversary knows the position of the secret blocks and of the known blocks in the stream, he can quickly see which collisions are useful, and recover the secret values by xoring with the known plaintext.

Attack Complexity. Let us denote the known fraction of the data as α and the secret and valuable fraction of the traffic as β (with fraction $1 - \alpha - \beta$ that is neither known nor valuable to the attacker). In order to recover some secret information, an attacker must collect roughly $1/2\alpha\beta$ collisions, so that one collision is between a valuable secret block and a known block. Following the analysis of Section 2.2, this requires about $\sqrt{1/\alpha\beta} \cdot 2^{n/2}$ blocks of data.

For instance, with $n = 64$, in an optimal case for the adversary half of the traffic is known, and half of the traffic is highly valuable ($\alpha = \beta = 1/2$). In this case, a collision an attack requires about:

$$\sqrt{1/\alpha\beta} \cdot 2^{n/2} = 2 \cdot 2^{n/2} = 2^{33}$$

blocks of data, which corresponds to just 64 GB. For a more concrete scenario, let us assume that the messages are HTTP queries of 512 bytes (64 blocs), with a secret 8-byte cookie (1 block), and that the remaining of the message is known, *i.e.* $\alpha = 63/64 \approx 1$, $\beta = 1/64$. The number of blocks needed by an attacker is roughly:

$$\sqrt{1/\alpha\beta} \cdot 2^{n/2} \approx 8 \cdot 2^{n/2} = 2^{35}$$

which correspond to 256 GB.

Rekeying at the Birthday Bound. The attack can also be carried out when rekeying is used, if the keys are changed with low frequency. In particular, even if the keys are changed before $2^{n/2}$ blocks are encrypted, there is still a non-negligible probability that a collision occurs.

More precisely, let us assume that keys are rotated after encrypting 2^r blocks ($r \leq n/2$). As explained in Section 2.2, the probability to have a collision between 2^r blocks is at least 2^{2r-n-2}. Therefore, an attacker needs $1/2\alpha\beta \cdot 2^{n+2-2r}$ sessions (*i.e.* $1/2\alpha\beta \cdot 2^{n+2-r}$ blocks) to collect an interesting collision. In practice, the attacker would collect all the data in a session, sort it to find collisions, and can discard the data before collecting data from the next session.

In an optimal setting for the attacker ($\alpha = \beta = 1/2$), if the keys are changed after $2^{n/2}$ blocks as recommended by ISO [21] ($r = n/2$), the number of blocks needed by an attacker is roughly:

$$1/2\alpha\beta \cdot 2^{n+2-r} = 2 \cdot 2^{n/2+2} = 2^{35}$$

which correspond to 256 TB. In the HTTP example above, the number of blocks needed is roughly:

$$1/2\alpha\beta \cdot 2^{n+2-r} \approx 32 \cdot 2^{n/2+2} = 2^{39}$$

which correspond to 4 TB. Therefore rekeying just before $2^{n/2}$ blocks only has a limited impact on collision attacks.

Optimal Rekeying Strategies. The optimal use of rekeying has been studied by Abdalla and Bellare in [1], with the goal of increasing the *encryption threshold* (the number of blocks that can be encrypted before the advantage of an adversary becomes significant). Their result depend on the key size κ and the block size n. When $n = \kappa$, the optimal strategy is to rekey after $2^{n/3}$ blocks, leading to an encryption threshold of $2^{2n/3}$ (rather than $2^{n/2}$ without rekeying). Concrete figures for AES-based modes are also given in [24].

On the other hand, when $\kappa \geq 2n$ (as is the case for 3DES and Blowfish-128), the optimal strategy is to rekey for every block. Assuming a perfect rekeying function, this leads to an encryption threshold of 2^n (rather than $2^{n/2}$). In actual protocols such as TLS or OpenVPN, rekeying requires a new key exchange, which can be quite expensive. Therefore, rekeying should probably be implemented with a fixed limit, in the order of a few megabytes. We can compute the *encryption threshold* when rekeying every 2^r blocks as 2^{n-r} (this figure correspond to the optimal case for an adversary, with $\alpha = \beta = 1/2$). For example, by rekeying every megabyte, a client and server can exchange a petabyte of data before collision attacks become significant.

Exploiting Repeated Messages. We now assume that a fixed message of 2^u blocks is sent repeatedly (this will be the case for the concrete attacks we describe in Sections 4.4 and 4.5). The attacker's goal is to recover the full message, assuming that $\alpha \cdot 2^u$ blocks are known or predictable.

If there is a single unknown block, we can use the previous analysis with $\alpha \approx 1$ and $\beta = 2^{-u}$: an attacker need roughly $2^{(n+u)/2}$ blocks. Indeed, this should lead to roughly 2^{u-1} collisions, and there is a good chance that one of these collision is between the secret block and a known block. For a more accurate analysis, we can use the results of van Oorschot and Wiener [31], which gives an expected complexity of $\sqrt{\pi/2} \cdot 2^{(n+u-1)/2}$ blocks to find 2^{u-1} collisions.

If there are a few unknown blocks ($k = (1 - \alpha) \cdot 2^u$), the number of encrypted blocks required to recover them can be approximated as:

$$\sqrt{H_k} \cdot \sqrt{\pi/2} \cdot 2^{(n+u-1)/2},$$

because the number of collision required follows the harmonic numbers [1] $H_k = \sum_{i=1}^{k} 1/i$.

On the other hand, if α is small, the attacker can use collisions to derive equations between the plaintext blocks, even if both blocks are unknown. Each collision reveals the xor between two plaintext blocks, which can seen as an edge in a graph whose vertexes are the plaintext blocks. If we model the graph as a random graph with 2^u vertexes, we know that with high probability, the graph will be fully connected when there are more than $\Theta(u/2 \cdot 2^u)$ edges [17]. At this point, all plaintext blocks can be recovered from a single known block. Therefore, the expected attack complexity becomes:

$$\sqrt{u} \cdot \sqrt{\pi/2} \cdot 2^{(n+u-1)/2}$$

Summary of Attack Scenario. With a fixed message of size 2^u repeatedly encrypted, an attacker that knows a fraction α of the message can recover the k missing blocks of plaintext ($k = (1 - \alpha) \cdot 2^u$) by observing about $\log(k) \cdot 2^{u-1}$ collisions. This gives an attack with a data complexity of

$$O\big(\sqrt{\log(k)} \cdot 2^{(n+u-1)/2}\big)$$

encrypted blocks. With rekeying every 2^r blocks ($r < n/2$), the complexity becomes:

$$O\big(\log(k) \cdot 2^{u+n+2-r}\big).$$

[1] Assuming that the time to recover each block follows a geometric law

Protocol	Year	Block ciphers	Mandatory	Rekey
TLS 1.0	1999	**3DES, DES, IDEA**	**3DES**	-
TLS 1.1	2006	AES, **3DES, DES**	**3DES**	2^{78}
TLS 1.2	2008	AES, **3DES**	AES	2^{78}
SSH 1	1995	**3DES, DES, IDEA**	**3DES**	-
SSH 2	2006	AES, **3DES, Blowfish**	**3DES**	2^{30},
IKEv1	1998	**3DES, DES, Blowfish**	**DES**	-
IKEv2	2010	AES, **3DES, Blowfish**	**3DES**	-
IPsec	2014	AES, **3DES**	AES	1GB

Table 1: Block ciphers in popular versions of Internet protocols; 64-bit ciphers are shown in red

In particular, we will apply this attack to break the security of encrypted HTTP connections when an authentication token is send with every request. We use the Beastly attacker of Figure 1 to generate a large number of requests, where the headers are mostly predictable or even controlled by the attacker ($\alpha \approx 1$), who can then recover the authentication token by observing collisions in the encrypted traffic.

3. 64-BIT BLOCK CIPHER USAGE ON THE INTERNET

Many of the most influential Internet security protocols, such as TLS, SSH, and IPsec were standardized at a time when 64-bit block ciphers, such as 3DES and Blowfish, were still considered strong. Consequently, these ciphers are still widely supported on the Internet. In this section, we investigate the real-world usage of these ciphers in order to quantify the feasibility and practical impact of our attacks.

3.1 Block Ciphers in Internet Standards

Table 1 summarizes the support for various block ciphers in several versions of the TLS, SSH, and IPsec standards. The table shows that protocols that were standardized before 2000 typically supported only 64-bit ciphers (primarily 3DES), although later protocol extensions enabled 128-bit ciphers (primarily AES). Still, old versions of these protocols remain quite popular on the web. Notably, each standard specifies a *mandatory* cipher that "MUST" be supported for interoperability, and in most of these specifications, the mandatory encryption algorithm is 3DES. This means that all implementations of TLS, SSH, and IPsec must implement 3DES, and unless the user explicitly disables it, they will offer to negotiate 3DES with their peers.

For example, in TLS 1.0 and 1.1, 3DES is the mandatory encryption algorithm, so all TLS libraries implement it and a vast majority of web servers support it. Although TLS 1.2 changed the mandatory cipher to AES, it still explicitly recommends 3DES as secure [13, Sec. 7.3]:

> If you negotiate 3DES with a 1024-bit RSA key exchange with a host whose certificate you have verified, you can expect to be that secure.

As a result, mainstream TLS libraries still treat 3DES to be at the same security level as AES-128. For example, until we disclosed the attacks in this paper, OpenSSL included 3DES ciphersuites in its HIGH-security list (it has now been moved to MEDIUM). Moreover, while the use of RC4 in TLS has been explicitly deprecated in RFC7465 in response to recent attacks, 3DES continues to be trusted by TLS libraries. In fact, it has even started being recommended as the new backup cipher in place of RC4.[2] TLS does not mandate rekeying for short block ciphers, and so TLS libraries leave the decision to rekey to their applications.

In SSH version 1, the strongest available ciphers are all 64-bit block ciphers: 3DES, Blowfish, and IDEA, and although SSH 2 introduced AES-based ciphers, it still labeled 3DES as the mandatory cipher. Consequently, popular implementations such as OpenSSH still use 3DES as the default cipher for SSH-1 connections, and 3DES continues to be the most widely supports cipher in SSH.[3] SSH standards recommend that implementations rekey after every 1GB of data, or after every $2^{n/4}$ blocks [8], but many popular SSH clients do not implement this feature.

The early IPSec specifications also widely used and recommended 3DES. For example, RFC4835 recommended 3DES as a mandatory (MUST) algorithm until 2007, although the more recent RFC7321 has downgraded it to a "MAY". Consequently, most IPSec libraries continue to support 3DES, and since IPSec deployments tend to be stable over a long period of time within corporate networks, we anticipate that the algorithm may continue to be supported on IPSec networks for the foreseeable future.

In summary, short block ciphers and 3DES in particular have widespread support in major secure transport protocols, making them vulnerable to the attacks described in this paper. However, to evaluate the real-world impact of this vulnerability, the key question is whether these ciphers are actually used in connections between popular clients and servers. In the rest of this section, we attempt to quantify this usage.

3.2 3DES and Blowfish usage in VPNs

TLS, SSH, and IPSec are often used to implement *tunnels* or virtual private networks (VPNs) across untrusted network connections. We have already discussed the widespread support for 64-bit block ciphers in these protocols, but estimating whether these ciphers are actually negotiated in practice is hard, because unlike web servers, most VPN services are not publicly accessible, and each VPN server may use its own configuration. However, we found a few VPN solutions that use 64-bit ciphers by default.

IPsec. Most IPSec-based VPN clients support 3DES for interoperability. Notably, some versions of Microsoft's L2TP VPN client use 3DES by default[4]; some versions of MAC OS X negotiate 3DES with certain VPN servers[5]; and some versions of CISCO IPsec VPN devices implement hardware optimizations for 3DES and consequently prefer it over other ciphers.[6]

OpenVPN. OpenVPN is a popular open-source VPN solution originally written by James Yonan. The default encryption for the transport protocol of OpenVPN is Blowfish, a 64-bit cipher, in CBC mode. OpenVPN supports two different ways of generating session keys to encrypt the messages. In pre-shared-key mode, static keys are used for all the traf-

[2] https://wiki.mozilla.org/Security/Server_Side_TLS
[3] https://blog.binaryedge.io/2015/11/10/ssh/
[4] https://support.microsoft.com/en-us/kb/325158
[5] http://www.jacco2.dds.nl/networking/openswan-macosx.html
[6] http://www.cisco.com/c/en/us/td/docs/ios/12_2/12_2z/12_2zj/feature/guide/gtaimvpn.html

Firefox Version				AES	
#	Release date	RC4	3DES	CBC	GCM
35	2015/01/13	20.4	0.15	27.5	51.9
36	2015/02/24	0.20	1.81	45.5	52.5
37	2015/03/31	0.13	1.81	41.7	56.4
38	2015/05/12	0.09	1.82	36.0	62.1
39	2015/07/02	0.08	1.81	31.6	66.5
40	2015/08/11	0.08	1.68	29.7	68.6
41	2015/09/22	0.08	1.44	27.9	70.6
42	2015/11/03	0.08	1.26	27.9	70.7
43	2015/12/15	0.05	1.19	28.6	70.2
44	2016/01/26	–	1.17	28.3	70.5
45	2016/03/08	–	1.13	28.1	70.8

Table 2: Cipher use for TLS connection from the Firefox web browser (percentage). The figures come from telemetry data available at https://telemetry.mozilla.org/ (we use the `SSL_SYMMETRIC_CIPHER_FULL` metric), and are gathered during the lifetime of a particular version. RC4 is fallback-only since version 36.

fic. In particular, there is no limit to the lifetime of those keys. In TLS mode, session keys are generated with a TLS handshake, using certificates to authenticate the peers. The session keys are updated periodically, with limits on the number of packets, the number of bytes, or a session time. The default configuration rekeys the tunnel every hour. In addition, the tunnel is rekeyed shortly before 2^{32} packets in TLS mode, because the packet counter is a 32-bit integer. In pre-shared-key mode, the packet counter is a 64-bit integer.

3.3 3DES usage in HTTPS

3DES is the second most widely supported cipher (after AES) in HTTPS servers, with about 87% of servers supporting it.[7] Furthermore, all popular web browsers support 3DES. The cipher that is actually negotiated for a TLS connection is chosen by the server, based on its local preference order and on the order in which the client advertises its ciphersuites. Since most modern browsers and servers prefer AES over 3DES, we find evidence that 1-2% of all TLS connections likely use 3DES in CBC mode, as detailed below.

Browser Telemetry. Telemetry data from Mozilla Firefox (given in Table 2) show that 3DES is used for more than 1% of HTTPS connections from Firefox browsers. Interestingly, the use of 3DES with Firefox has actually increased recently, as a consequence of the removal of RC4 from the list of supported ciphers in version 36 (following [3] and RFC 7465). Indeed a number of servers are configured to use, in order of preference, first RC4, then 3DES, and now use 3DES with Firefox.

Other browser vendors have privately reported the following telemetry data to us. Microsoft sees 1.2% 3DES connections for Internet Explorer and Edge, but only 0.8% system wide (the lower value can be explained by the fact that some applications still support RC4). Google Chrome sees 1.1% of connections using 3DES, but this number peaked at 1.6% soon after RC4 was disabled.

[7] https://securitypitfalls.wordpress.com/2016/04/17/february-2016-scan-results-incomplete/

3DES support	Andr5	FF44	IE11	Chr47
Top 1k	93%	0.4%	1.6%	1.6%	1.6%
Top 10k	92%	1.0%	2.1%	2.1%	2.1%
Top 100k	89%	0.8%	1.9%	1.9%	1.9%
Top 1m	86%	0.6%	1.3%	1.3%	1.3%

Table 3: 3DES support for HTTPS servers in Alexa's top website list. We count servers that support 3DES, and servers that would select 3DES with common browsers: Android 5.0 integrated browser, Firefox 44, Internet Explorer 11, and Chrome 47. Scans performed on February 16 – 18, 2016.

Scanning the Top 1M Websites. We performed a scan of the top 1 million servers as listed by Alexa using the cipherscan tool[8]. We found that 86% of the servers that support TLS include 3DES as one of the supported ciphers. Moreover, using the ciphersuites sent by a modern client (Firefox 44, IE 11 or Chrome 47), we estimate that 1.2% will actually use a 3DES based ciphersuite. We report detailed figures in Table 3.

Windows XP Clients and Windows 2003 Servers. The Windows XP and Windows Server 2003 operating systems do not support AES-based ciphersuites, although support for AES can be added with an optional hotfix. With the default configuration, they support only RC4, 3DES, DES, and RC2-40. While these operating systems are not supported anymore by Microsoft, they still have users, and this creates situations where the best available cipher is 3DES.

If a client uses Internet Explorer on Windows XP or Windows Server 2003 to connect to a modern web server (with no RC4 ciphersuites), the connection will use 3DES or fail. Indeed, supporting Windows XP clients is the main stated reason for servers to continue supporting RC4 and 3DES.

If a web server runs on IIS 6.0 (*i.e.* Windows XP or Windows Server 2003), it will offer only RC4 and 3DES ciphersuites, unless support for AES ciphersuites has been added. If a modern web browser visits the website over HTTPS, the connection will use 3DES. According to web server surveys[9], IIS 6.0 powers about 1.9% of SSL enabled web servers, but it is likely used even more within corporate networks, and these servers would not appear in public scans.

Long-lived HTTPS connections. In order to mount the attack in practice, we need to find client and servers that not only negotiate the use of 3DES, but also exchange a large number of HTTP request in the same TLS connection (without rekeying). This is possible using a persistent HTTP connection, as defined in HTTP/1.1 (Keep-Alive).

On the client side, all browsers that we tested (Firefox, Chrome, Opera) will reuse a TLS connection as long as the server keeps it open. On the server side, we found that a number of HTTP servers will close the TLS connection even when it is still active. In particular, Apache and Nginx limit the number of requests sent in the same connection, with a maximum of 100 in the default configuration[10]. On

[8] https://github.com/jvehent/cipherscan

[9] http://www.securityspace.com/s_survey/sdata/201604/servers.html

[10] The setting is called `MaxKeepAliveRequests` for Apache, and `keepalive_requests` for Nginx.

Website	connection closed after
google.com	1 hour†
facebook.com	no limit detected†
youtube.com	no limit detected†
yahoo.com	no limit detected†
live.com	up to one day†
baidu.com	100 requests
amazon.com	100 requests
wikipedia.org	100 requests
qq.com	2000 requests
twitter.com	50 requests

Table 4: Maximum connection lifetime for Alexa's top 10 servers (detected experimentally). The "†" symbol marks servers which accept at least one million requests in a single TLS connection.

the other hand, IIS does not seem to have such a limit. In practice, many high profile servers accept a very large number of requests in a single TLS connection. We made some experiments using a simple Perl script to send simple HTTP requests over a TLS connection, using pipelining to maximize the throughput. As seen in Table 4, we found that about half of the servers in Alexa's top 10 support a large number of requests without rekeying.

For a better estimate of the number of vulnerable servers, we tested servers from Alexa's top 10k that negotiate 3DES with a modern client. We identified 11483 different HTTPS servers[11], and found that 226 of them (1.9%) negotiate 3DES with a modern client. Moreover, 72 of these (0.6% of the total) also accept to keep a connection open for at least 800k requests. Consequently, the duration of the attack is not unrealistic, at least from the viewpoint of browsers and servers, and we estimate that at least 0.6% of HTTPS connections are vulnerable to our attacks.

Downgrading to 3DES. From our measurements, only 1-2% of HTTPS connections use 3DES, and perhaps even fewer of these connections are to servers that allow long-lived TLS connections. However, ignoring the danger posed by 3DES would be dangerous for several reasons. 3DES is typically the second or third most preferred encryption algorithms in HTTPS, so any change in the client or server configurations, say in response to some future attack on AES, would cause a large number of connections to fallback to 3DES. Furthermore, as recent downgrade attacks on TLS (e.g. FREAK) have demonstrated, protocol and implementation flaws can sometimes cause a weaker cipher to be negotiated. For example, if a downgrade attack from TLS 1.2 to SSL 3 becomes possible, then the strongest SSL 3 cipher that can be negotiated may well be 3DES. As an example, we describe a downgrade attack to 3DES that is currently possible on TLS connections. Although it does not enable our full attack, it nevertheless serves as a useful warning for practitioners.

Downgrading False Start. TLS False Start [23] is an optimization to the TLS handshake that reduces its latency. A client implementing False Start will start sending data before receiving the Finished message from the server. This allows an attacker to force the selection of a ciphersuite by tampering with the ServerHello message (he can replace AES by 3DES, for example) and the client will start sending data before verifying that the ciphersuite was really selected by the authentic server. If the client is willing to send millions of requests and gigabytes of data before finishing the handshake, this could allow our attack even against a server that would not select a 3DES ciphersuite. While 3DES ciphersuite are not recommended for use with False Start in the RFC draft [23], it turns out that they are white-listed for use with False Start with Internet Explorer. We have informed the Microsoft security team and they decided to remove 3DES from this white-list.

[11]We tested common sub-domains: `www`, `m`, `secure`, `signin`, `auth`, `account`, `login`

4. ATTACKING AUTHENTICATED HTTP OVER TLS AND OPENVPN

We now demonstrate concrete attacks against authenticated HTTP sessions even when they are secured by TLS or OpenVPN. First, we identify a few examples of secret authentication credentials that are repeatedly sent by the browser on every request. We then show how we can recover these secrets using block cipher collisions.

4.1 HTTP bearer tokens

Cookie-based Sessions. Modern HTTPS websites use a variety of methods to manage authenticated sessions with their clients. The most popular mechanism is secure *cookies* as specified in RFC6265. Once a user has logged in, the server sets a cookie containing a secret value on the user's browser. The browser will then send the cookie on all subsequent requests to the website, implicitly authenticating the user. For example, if domain.com sets a cookie `C=XXXX`, a subsequent HTTPS request from the browser looks like:

```
GET /path/to/file HTTP/1.1
User-Agent: Mozilla/4.0 ...
Host: domain.com
Cookie: C=XXXXX
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

Cookies are sensitive, because an attacker who obtains a session cookie can then log in as the user from a different browser. The cookie acts as a *bearer token* that carries the user's delegated credential. Except for its short-lived nature, it has the same security requirements as a password. Indeed, many websites, such as Facebook, use cookies with very long expiry times, making cookie theft even more attractive.

A cookie for a website is normally included in all requests to that website whether that request was initiated by the user, or a script on the website, or even by a different website. Browsers sometimes impose stricter rules for this last category of requests, called cross-origin or cross-domain requests. For example, XmlHttpRequests sent from one domain to another may not have cookies attached. However, cross-domain requests for images or iframes will still send cookies.

HTTP BasicAuth. On corporate networks and even on the web, many websites are protected using the HTTP Basic Authentication mechanism specified in RFC7617. When the server asks for Basic authentication, the browser prompts the user to enter a username and password into a special dialog and then sends this information (in plaintext) as an HTTP header of the form:

```
Authorization: Basic dGVzdDoxMjPCow=
```

Once a user has entered his login information once, the browser will typically cache this information and use it on all subsequent requests to the server. Notably, even if a different website creates a cross-domain request to the authenticated server, the BasicAuth credentials will be automatically sent by the browser.

BasicAuth credentials contain the user's password and are hence security-critical. One should only use this authentication mechanism over HTTPS but a number of corporate websites use BasicAuth over HTTP, under the assumption that their users can only access the website over a VPN or some other secure connection.

OAuth and OpenID Connect Access Tokens. With the growth in popularity of single sign-on protocols, many websites now allow users to login with third-party credentials, such as OAuth or OpenID Connect tokens, issued by major websites like Google and Facebook. These tokens are effectively bearer tokens that must then be attached to all authentication and authorization requests and are sent either within the URI or in an Authorization header. While the use of these tokens is specific to each website, an attacker can usually repeatedly trigger the sending of these tokens by re-initiating the login sequence. An attacker who steals one of these tokens can then impersonate the user as well as retrieve sensitive data about the user from the login provider.

4.2 The Beastly Attack Scenario

Our attack scenario, depicted in Figure 1, is similar to the setup used in recent attacks on RC4 [3, 19, 32]. The attacker wants to steal some bearer token that is being repeatedly sent by a browser to a website secured with HTTPS, or an HTTP website accessed through a VPN. We assume that the attacker can control some JavaScript on a web page loaded by the user's browser, either by actively tampering with an HTTP response on the wire, or by hosting a malicious website that the user is fooled into visiting. We also assume that the attacker can observe all the encrypted traffic between the target browser and the secure website.

We further assume that the data is encrypted with a 64-bit block cipher in CBC mode (either an HTTPS connection where the client and server have negotiated 3DES, or an HTTP connection though a VPN encrypted with Blowfish or 3DES). Suppose the victim is already logged in to a website and has a session cookie. The attacker runs malicious JavaScript code on the victim's browser that repeatedly sends HTTP queries to the target website server, each containing the session cookie. If he sends close to 2^{32} queries, a collision is expected between a ciphertext block corresponding to the cookie (c_i), and a known block (c_j), containing a known part of the query. The collision attack against CBC reveals the session cookie: $p_i = p_j \oplus c_{i-1} \oplus c_{j-1}$.

Note that most of the traffic generated by the attacker is known or predictable. The requested URL is chosen by the attacker, and all the headers excepted the cookie are predictable and can be observed in any plaintext HTTP traffic. We will assume that the Cookie takes 16-bytes, *i.e.* two blocks, and that the attacker has aligned the cookie to a block boundary using other headers. As explained in Section 2.2, if a query contains 2^u known blocks, the number of encrypted blocks required for the attack is about:

$$\sqrt{H_2} \cdot \sqrt{\pi/2} \cdot 2^{(n+u-1)/2}.$$

In practice, an HTTP query is about 512 bytes (64 blocks), which implies a total complexity of $2^{35.1}$ blocks ($2^{29.1}$ queries), and a total captured trace of size 280 GB.

However, the attack can be made faster by using larger queries: this increases the total amount of data to capture, but it reduces the total number of queries. Since the speed of clients and servers is dependent on the number of queries, a larger query size can result in a faster attack overall. In our experiments, we got good results with 4 kB queries (512 blocks). In this setting, the attack requires $2^{36.6}$ blocks (785 GB), but this corresponds to only $2^{27.6}$ queries: increasing the requests size by a factor 8 (from 512 bytes to 4 kB) increases the total data to capture by a factor $\sqrt{8}$, and reduces the number of queries needed by a factor $\sqrt{8}$. In practice large queries can be produced by adding a long query string to the URL.

4.3 Proof-of-concept Code

The attacker code consists of two parts: a JavaScript program that sends a large number of HTTP requests, and a network adversary who processes the resulting ciphertext to recovers a 16-byte secret.

Man-in-the-browser code. We experimented with hidden `<img>` tags in a web-page with Javascript code to the `onload` event to reload the images repeatedly, and web workers issuing `XmlHttpRequests`. Cookies are automatically inserted when loading images from a different domain, and are also included in cross-domain `XmlHttpRequests` if the `withCredentials` property is set to `true` (this is only allowed for synchronous request, and the request result can not be read, but these restrictions don't prevent our attack). According to our experiments, request are generated faster using `XmlHttpRequest`. In order to avoid caching, we request a non-cacheable resource. Alternatively, we can include a unique query string in each URL, but the responses are still cached by the client and this can affect the query rate. We experimented with several browsers, and we obtained the best results with Firefox Developer Edition 47.0a2 [12].

Recovering collisions. We captured the encrypted packets with `tcpdump` and used a C++ program to extract the ciphertext blocks (using `libpcap`). In both the HTTPS attack and the OpenVPN attack, each HTTP query is sent in a separate encrypted record, which contain the plaintext at a fixed position, as well as some extra information (packet number, padding, MAC, ...). Therefore, it is easy to know to which plaintext block corresponds each ciphertext block, and to align the cookie to a block limit.

After capturing all the traffic, the C++ program sorts the ciphertext blocks in order to locate collisions. Since the amount of data is quite large (hundreds of gigabytes), we use the external sort implementation of the `stxxl` library. With a NAS storage, sorting the data took around four hours.

4.4 Attacking Basic Auth over OpenVPN

To demonstrate the attack against OpenVPN, we use a pre-shared-key tunnel between two physical machines running Linux, with Firefox Developer Edition 47.0a2 on one side, and an nginx server on the other side. Access to the server is protected by BasicAuth, and the user has entered his

[12] We have similar results with the stable version Firefox 48.0 which was released after we made our experiments.

credentials. Using the default OpenVPN settings, the tunnel is encrypted with Blowfish in CBC mode.

We use the Javascript code described in the previous section to send a large number of requests to the server though the tunnel. We found that increasing the size of the request to 4 kB does not significantly reduce the query rate, but reduces the number of queries required for the attack. In our setting, the browser generates about 2900 requests per second, using several web Worker running in parallel. We expect the first collision after roughly $2^{32.3}$ blocks (40 GB), *i.e.* one hour. In practice, we detected the first collision early, after only 30 minutes ($2^{31.3}$ blocks); as expected, the predicted xor difference was correct. The full attack needs about $2^{36.6}$ blocks (785 GB) to recover a two-block secret out of 4 kB messages; this should take about 19 hours in this setting. In our demo, it took 18.6 hours and 705 GB, and we successfully recovered the 16-byte authentication token.

OpenVPN in TLS mode. In TLS mode, OpenVPN rekeys the tunnel every hour with the default settings (as apposed to the pre-shared-key mode used above that doesn't do any rekeying). With our experimental setting, this means we can only capture about $2^{23.3}$ requests encrypted with a given key, *i.e.* $2^{32.3}$ blocks. Therefore, we expect to detect a collision with probability close[13] to 1/2. In order to recover a two-byte secret out of 4 kB messages, we need about 384 collisions as explained in Section 2.2. Therefore, this attack will take roughly 768 hours, or 32 days.

This illustrate the point that keys should be changed *well before* reaching 2^{32} blocks: if each session is broken after 2^{32} blocks, there is still a collision with probability roughly $1-e^{-1/2} \approx 0.4$. This increases the time required by the attack to 40 days, but it will still succeed with high probability.

4.5 Attacking Secure Cookies over HTTPS

The attack against HTTPS connection is very similar to the attack against OpenVPN, as long as we have a client and server that negotiate 3DES, and that keep a connection open for a long time.

Vulnerable Websites. As detailed in Section 3.3, about 0.6% of the top Alexa 100K websites support 3DES and allow long-lived HTTPS connections. This list contains many high-profile e-commerce and banking websites, including the ones shown in Table 5. In addition, we found a number of domains were the server serving `domain.com` will use 3DES, but it just redirects the client to a better configured server for `www.domain.com`. This could still open the way for our attacks, if sessioncookies for `www.domain.com` are set at the top-level domain and hence also sent to `domain.tld`. The last three websites in Table 5 belong to this category.

Proof-of-Concept Attack Demo. In our setup we use Firefox Developer Edition 47.0a2 running on Linux and an IIS 6.0 server in a Windows Server 2003 R2 SP2 Virtual machine. The default configuration of IIS 6.0 with all recommended updates offers only RC4 and 3DES ciphers, and will use 3DES with Firefox and other recent browsers where RC4 is disabled. Moreover, IIS 6.0 supports HTTP/1.1 and keeps an active connection open for an arbitrary long time.

We use the Javascript code described in Section 4.3, but we note that with several `Workers` running in parallel (or several `<img>` tags), modern browsers open a few parallel connections to the server (typically, 6) and split the requests over these connections. For our attack, we need to maximize the throughput over a *single* connection. This can be achieved by disturbing some of the connections so that most of request are send in a single. In our setup, we used `iptables` rules to limit the rate of all connections except one. In a real attack, this would be done by an active man-in-the-middle, but a passive man-in-the-middle can also mount the attack – it will just take more time to collect the data.

[13]The probability can be computed as $1 - e^{-2^{32.2}2^{32.2}/2^{64}}$.

Website	Category
https://signin.ebay.com	E-commerce
https://account.nasdaq.com	Finance
https://www.bancomercantil.com	Banking
https://www.unionbankonline.co.in	Banking
https://ziraatbank.com.tr	Banking
https://www.state.nj.us	Government
https://secure.match.com	Dating
https://amadeus.net	Travel
https://walmart.com	Corporate
https://citrix.com	Corporate

Table 5: High-value websites that negotiate 3DES with modern browsers, and accept at least 1 million requests in the same TLS session

On Firefox Developer Edition 47.0a2, with a few dozen `Worker`s running in parallel, we can send up to 2000 requests per second in a single TLS connection. To further reduce the time needed for the attack, we inject a padding cookie to expand the requests to 4 KB (512 blocks). In our setting this reduces the rate to 1500 requests per second, but it still leads to a faster attack. Again, we expect the first collision after roughly $2^{32.3}$ blocks (40 GB), *i.e.* $2^{23.3}$ queries. This should take slightly less than two hours.

In our experiment, we were lucky to detect the first collision after only 25 minutes ($2^{20.1}$ requests), and we verified that the collision revealed the xor of two plaintexts blocks. As seen previously, the full attack should require $2^{36.6}$ blocks (785 GB) to recover a two-block cookie, which should take 38 hours in our setting. Experimentally, we recovered the two-block cookie after only 30.5 hours and 610 GB.

Note that special care must be taken to recover the first block of a TLS fragment. When a collision involves the first ciphertext block ($c_i^{(j)} = c_0^{(j')}$), this gives an equation involving the IV ($c_{i-1}^{(j)} = c_{-1}^{(j')}$). In particular, this requires special treatment for TLS 1.0, because the IV of a given of a given record is not explicitly included in the message: it is the last ciphertext block of the previous record.

Our attack above relied on knowing most of the plaintext, but if we have enough data we can even recover most of the plaintext using just a single known block. In our experiment, after 65 hours ($2^{28.3}$ requests, 1.3TB) we can recover 492 blocks out of 510, assuming a single known plaintext block.

5. IMPACT AND MITIGATION

We have demonstrated the first concrete attacks on mainstream Internet protocols that exploit block cipher collisions. Our attacks can recover valuable secrets such as HTTP cookies and passwords in under 40 hours. Our attacks impact a

majority of OpenVPN connections and an estimated 0.6% of HTTPS connections to popular websites. We expect that our attacks also impact a number of SSH and IPsec connections, but we do not have concrete measurements for these protocols. Like many recent attacks on TLS, such a BEAST and RC4 NOMORE, the underlying principles behind our attacks were well known to cryptographers. Our goal is to raise awareness among practitioners about the vulnerabilities of short block ciphers and on safe ways of using them.

5.1 Comparison with RC4 attacks

Our attack scenario is very similar to the setup of the recent attacks on the use of RC4 in HTTPS. We use the same man-in-the-browser setting to generate a large number of HTTP requests, and the data complexity of our attacks is comparable to these prior works:

- the first attack by Al Fardan *et al.* [3] uses between 2^{28} and 2^{32} sessions, with rekeying for each session, to recover the first 220 bytes a the message (but the cookie is not usually in these 220 bytes);
- the second attack by Al Fardan *et al.* [3] uses between 2^{33} and 2^{34} request, in the same TLS session or in different sessions, to recover a cookie;
- Garman *et al.* [19] improved the first attack to require only 2^{26} sessions when targeting a BasicAuth password, using a better guessing strategy;
- the latest attack by Vanhoef and Piessens [32] requires $2^{30.2}$ requests, in the same TLS session or in different sessions, to recover a cookie.

In comparison, our attack requires only $2^{29.1}$ short queries of 512 bytes (280 GB in total), which can be reduced to $2^{27.6}$ longer queries of 4 kB (785 GB in total). However, these numbers are for the case when all the data is encrypted within the same session. Even if the amount of data sent on a single connection is limited, as long as the limit is close enough to the birthday bound, we can still mount our attacks across multiple parallel and sequential sessions, albeit with a higher data and time complexity.

5.2 Mitigation

The obvious way to avoid these attacks is to stop using legacy 64-bit block-ciphers. There is no good reason to configure a VPN or an HTTPS server to prefer 3DES (or Blowfish) over AES. AES is more secure, is a FIPS standard, and will almost certainly be faster: it is implemented in hardware in most server and desktop CPUs, and can be implemented efficiently in dedicated hardware when an extremely high throughput is required.

On the other hand, there are specific uses of cryptography were 128-bit block ciphers can not be used, because of legacy reasons, or due to the relatively large hardware footprint of 128-bit ciphers (in this case, *lightweight* 64-bit block ciphers such as PRESENT or HIGHT would typically be used). In these scenarios, protocol designers should pay close attention to the issue of birthday attacks. In particular, they should verify that the amount of data encrypted with a fixed key is *significantly* smaller than 2^{32} blocks, or use modes that provide security beyond the birthday bound, such as CENC [22] or PMAC_plus [33].

Concretely, we recommend the following measures to prevent our attack:

1. Web servers and VPNs should be configured to prefer 128-bit ciphers. According to our scans, about 1.1% of the top 100k web server from Alexa, and 0.5% of the top 1 million, support AES but prefer to use 3DES.

2. Web browsers should offer 3DES as a fallback-only cipher, to avoid using it with servers that support AES but prefer 3DES.

3. TLS libraries and web browsers and servers should limit the length of TLS sessions with a 64-bit cipher. This could be done in TLS renegotiation, or in some cases by closing the connection and starting a new one (*i.e.* limiting HTTP/1.1 Keep-Alive, SPDY, and HTTP/2 with 3DES ciphersuites).

5.3 Responsible Disclosure

We have communicated our results and concerns to the OpenVPN team, and to various website owners, browser vendors, and TLS libraries. They all acknowledged the issue, and are working on implementing countermeasures. The TLS vulnerability received CVE number CVE-2016-2183.
OpenVPN will display a warning to users who choose to use 64-bit ciphers and encourage them to transition to AES-GCM (cipher negotiation is also being implemented in the 2.4 branch). It will also implement a default renegotiation limit of 64MB when used in TLS mode.
OpenSSL has moved 3DES ciphersuites from the `HIGH` category to `MEDIUM` in the 1.0.2 branch, and will disable it by default in the upcoming 1.1.0 release.
Akamai will offer an option for web server administrators to drop 3DES from the offered ciphers.
Apple has disabled 3DES on icloud.com and is recommending that all its customers disable 3DES on their websites.

Currently, most browsers see about 1% of their connections using 3DES, and vendors consider this number too high to simply disable 3DES on the client side, since too many websites would be broken. So, they are instead considering implementing data limits per connection to force rekeying, or offering 3DES ciphersuites only in a fallback negotiation if no AES ciphersuite is acceptable to the server.
Mozilla is implementing data limits for all ciphersuites: https://bugzilla.mozilla.org/show_bug.cgi?id=1268745 .
Microsoft has removed 3DES from the False Start whitelist: https://technet.microsoft.com/library/security/3155527.aspx

More details about implemented countermeasures will be added to our webpage as they become available:
http://sweet32.info

6. REFERENCES

[1] M. Abdalla and M. Bellare. Increasing the lifetime of a key: a comparative analysis of the security of re-keying techniques. In T. Okamoto, editor, *ASIACRYPT 2000*, volume 1976 of *LNCS*, pages 546–559. Springer, Heidelberg, Dec. 2000.

[2] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Z. Béguelin, and P. Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In I. Ray, N. Li, and C. Kruegel:, editors, *ACM CCS 15*, pages 5–17. ACM Press, Oct. 2015.

[3] N. J. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. N. Schuldt. On the Security of RC4 in TLS. In S. T. King, editor, *USENIX Security*, pages 305–320. USENIX Association, 2013.

[4] N. J. AlFardan and K. G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540. IEEE Computer Society Press, May 2013.

[5] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Kasper, S. Cohney, S. Engels, C. Paar, , and Y. Shavitt. DROWN: Breaking TLS using SSLv2, 2016. https://drownattack.com.

[6] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *38th FOCS*, pages 394–403. IEEE Computer Society Press, Oct. 1997.

[7] M. Bellare, J. Kilian, and P. Rogaway. The security of cipher block chaining. In Y. Desmedt, editor, *CRYPTO'94*, volume 839 of *LNCS*, pages 341–358. Springer, Heidelberg, Aug. 1994.

[8] M. Bellare, T. Kohno, and C. Namprempre. The Secure Shell (SSH) Transport Layer Encryption Modes. IETF RFC 4344, 2006.

[9] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *2015 IEEE Symposium on Security and Privacy*, pages 535–552. IEEE Computer Society Press, May 2015.

[10] K. Bhargavan and G. Leurent. Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH. In *ISOC Network and Distributed System Security Symposium (NDSS16)*, 2016.

[11] K. K. Bodo Moller, Thai Duong. This POODLE Bites: Exploiting The SSL 3.0 Fallback, 2014. https://www.openssl.org/~bodo/ssl-poodle.pdf.

[12] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In P. Paillier and I. Verbauwhede, editors, *CHES 2007*, volume 4727 of *LNCS*, pages 450–466. Springer, Heidelberg, Sept. 2007.

[13] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. IETF RFC 5246, 2008.

[14] W. Diffie and M. E. Hellman. Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE*, 67(3):397–427, 1979.

[15] T. Duong and J. Rizzo. Here come the ⊕ ninjas. *Unpublished manuscript*, 2011.

[16] M. Dworkin. Recommendation for Block Cipher Modes of Operation. NIST Special Publication 800-38{A,B,C,D}, National Institute for Standards and Technology, 2001 – 2011.

[17] P. Erdos and A. Rényi. On the evolution of random graphs. *Bull. Inst. Internat. Statist*, 38(4):343–347, 1961.

[18] S. Frankel and S. Krishnan. IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap. IETF RFC 6071, 2011.

[19] C. Garman, K. G. Paterson, and T. V. der Merwe. Attacks Only Get Better: Password Recovery Attacks Against RC4 in TLS. In J. Jung and T. Holz, editors, *USENIX Security*, pages 113–128. USENIX Association, 2015.

[20] D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B.-S. Koo, C. Lee, D. Chang, J. Lee, K. Jeong, H. Kim, J. Kim, and S. Chee. HIGHT: A new block cipher suitable for low-resource device. In L. Goubin and M. Matsui, editors, *CHES 2006*, volume 4249 of *LNCS*, pages 46–59. Springer, Heidelberg, Oct. 2006.

[21] On the Assessment of Cryptographic Techniques and Key Lengths, 4th edition. ISO/IEC JTC 1/SC 27 Standing Document 12, May 2014. Available online: http://www.din.de/blob/78392/6f4bbd95d0cf11d1b32784948039600b/sc27-sd12-data.pdf.

[22] T. Iwata. New blockcipher modes of operation with beyond the birthday bound security. In M. J. B. Robshaw, editor, *FSE 2006*, volume 4047 of *LNCS*, pages 310–327. Springer, Heidelberg, Mar. 2006.

[23] A. Langley, N. Modadugu, and B. Moeller. Transport Layer Security (TLS) False Start. Internet Draft, Nov. 2015. https://tools.ietf.org/html/draft-ietf-tls-falsestart-01.

[24] A. Luykx and K. G. Paterson. Limits on authenticated encryption use in TLS, march 2016. http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf.

[25] D. McGrew. Impossible plaintext cryptanalysis and probable-plaintext collision attacks of 64-bit block cipher modes. In *Fast Software Encryption Workshop (FSE)*, 2013. https://eprint.iacr.org/2012/623.

[26] D. McGrew and P. Hoffman. Cryptographic Algorithm Implementation Requirements and Usage Guidance for Encapsulating Security Payload (ESP) and Authentication Header (AH) . IETF RFC 7321, 2014.

[27] E. Petrank and C. Rackoff. CBC MAC for real-time data sources. *Journal of Cryptology*, 13(3):315–338, 2000.

[28] J. Rizzo and T. Duong. The crime attack. In *EKOparty Security Conference*, volume 2012, 2012.

[29] P. Rogaway. Problems with Proposed IP Cryptography. Unpublished draft, 1995. http://web.cs.ucdavis.edu/~rogaway/papers/draft-rogaway-ipsec-comments-00.txt.

[30] P. Rogaway. Evaluation of Some Blockcipher Modes of Operation. Technical report, CRYPREC, Feb 2011.

[31] P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.

[32] M. Vanhoef and F. Piessens. All Your Biases Belong to Us: Breaking RC4 in WPA-TKIP and TLS. In J. Jung and T. Holz, editors, *USENIX Security*, pages 97–112. USENIX Association, 2015.

[33] K. Yasuda. A new variant of PMAC: Beyond the birthday bound. In P. Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 596–609. Springer, Heidelberg, Aug. 2011.

[34] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. IETF RFC 4253, 2006.

The Missing Difference Problem, and its Applications to Counter Mode Encryption*

Gaëtan Leurent and Ferdinand Sibleyras

Inria, France
{gaetan.leurent,ferdinand.sibleyras}@inria.fr

Abstract. The counter mode (CTR) is a simple, efficient and widely used encryption mode using a block cipher. It comes with a security proof that guarantees no attacks up to the birthday bound (*i.e.* as long as the number of encrypted blocks σ satisfies $\sigma \ll 2^{n/2}$), and a matching attack that can distinguish plaintext/ciphertext pairs from random using about $2^{n/2}$ blocks of data.
The main goal of this paper is to study attacks against the counter mode beyond this simple distinguisher. We focus on message recovery attacks, with realistic assumptions about the capabilities of an adversary, and evaluate the full time complexity of the attacks rather than just the query complexity. Our main result is an attack to recover a block of message with complexity $\tilde{\mathcal{O}}(2^{n/2})$. This shows that the actual security of CTR is similar to that of CBC, where collision attacks are well known to reveal information about the message.
To achieve this result, we study a simple algorithmic problem related to the security of the CTR mode: the missing difference problem. We give efficient algorithms for this problem in two practically relevant cases: where the missing difference is known to be in some linear subspace, and when the amount of data is higher than strictly required.
As a further application, we show that the second algorithm can also be used to break some polynomial MACs such as GMAC and Poly1305, with a universal forgery attack with complexity $\tilde{\mathcal{O}}(2^{2n/3})$.

Keywords: Modes of operation, CTR, GCM, Poly1305, Cryptanalysis

1 Introduction

Block ciphers (such as DES or the AES) are probably the most widely used cryptographic primitives. Formally, a block cipher is just a keyed family of permutations over n-bit blocks, but when combined with a mode of operation, it can provide confidentiality (*e.g.* using CBC, or CTR), authenticity (*e.g.* using CBC-MAC, CMAC, or GMAC), or authenticated encryption (*e.g.* using GCM, CCM, or OCB). A mode of operation defines how to divide a message into blocks, and how to process the blocks one by one with some chaining rule.

The security of block ciphers is studied with cryptanalysis, with classical techniques such as differential [8] and linear [27] cryptanalysis, dedicated techniques like the SQUARE attack [9], and ad-hoc improvements for specific targets. This allows to evaluate the security margin of block ciphers, and today we have a high confidence that AES or Blowfish are as secure as a family of pseudo-random permutations with the same parameters (key size and block size).

On the other hand, modes of operation are mostly studied with security proofs, in order to determine conditions where using a particular mode of operation is safe. However, exceeding those conditions doesn't imply that there is an attack, and even when there is one, it can range from a weak distinguisher to a devastating key recovery. In order to get a better understanding of the security of modes of operations, we must combine lower bound on the security from security proofs, and upper bounds from attacks.

In particular, most of the modes used today are sensible to birthday attacks because of collisions; those attacks can even be practical with 64-bit block ciphers, as shown in [7], but security proofs don't tell us how dangerous the attacks are. For instance, the CBC and CTR modes have been proven secure against chosen plaintext attacks up to $\sigma \ll 2^{n/2}$ blocks of encrypted data [5,35]. Formally, the security statements bound the maximum advantage of an attacker against the modes as follows:

$$\mathbf{Adv}^{\mathrm{CPA}}_{\mathrm{CBC}-E} \leq \mathbf{Adv}^{\mathrm{prp}}_{E} + \sigma^2/2^n,$$
$$\mathbf{Adv}^{\mathrm{CPA}}_{\mathrm{CTR}-E} \leq \mathbf{Adv}^{\mathrm{prp}}_{E} + \sigma^2/2^{n+1}.$$

Both statement are essentially the same, and become moot when σ reaches $2^{n/2}$, but attacks can actually be quite different.

More precisely, the CBC mode is defined as $c_i = E(m_i \oplus c_{i-1})$, with E a block cipher. A collision between two ciphertext blocks $c_i = c_j$ is expected after $2^{n/2}$ blocks, and reveals the xor of two plaintext blocks: $m_i \oplus m_j = c_{i-1} \oplus c_{j-1}$. On the other hand, the counter mode is defined as $c_i = E(i) \oplus m_i$. There are no collisions in the inputs/outputs of E, but this can actually be used by a distinguisher. Indeed, if an adversary has access to $2^{n/2}$ known plaintext/ciphertext pairs, he can recover $E(i) = c_i \oplus m_i$ and detect that the values are unique (because E is a permutation), while collisions would be expected with a random ciphertext. Both attacks have the same complexity, and show that the corresponding proofs are tight. However, the loss of security is quite different: the attack against CBC lets an attacker recover message blocks from collisions (as shown in practice in [7]), but the attack against the counter mode hardly reveals any useful information.

In general, there is a folklore belief that the leakage of the CTR mode is not as bad as the leakage of the CBC mode. For instance, Ferguson, Schneier and Kohno wrote [15, Section 4.8.2] (in the context of a 128-bit block cipher):

> CTR leaks very little data. [...] It would be reasonable to limit the cipher mode to 2^{60} blocks, which allows you to encrypt 2^{64} bytes but restricts the leakage to a small fraction of a bit.
> When using CBC mode you should be a bit more restrictive. [...] We suggest limiting CBC encryption to 2^{32} blocks or so.

Our contribution. The main goal of this paper is to study attacks against the counter mode beyond the simple distinguisher given above. This is an important security issue, because uses of the CTR mode with 64-bit block ciphers could be attacked in practice. We consider generic attacks that work for any instance of the block cipher E, and assume that E behaves as a pseudo-random permutation. The complexity of the attacks will be determined by the block size n, rather than the key size, and we focus on the asymptotic complexity, using the Big-O notation $\mathcal{O}()$, and the Soft-O notation $\tilde{\mathcal{O}}()$ (ignoring logarithmic factors).

We consider message recovery attacks, where an attacker tries to recover secret information contained in the message, rather than recovering the encryption key k. Following recent attacks against HTTPS [12,2,7], we assume that a fixed message containing both known blocks and secret blocks is encrypted multiple times (this is common with web cookies, for instance). As shown by McGrew [28], this kind of attack against the CTR mode can be written as a simple algorithmic problem: the *missing difference problem*, defined as follows: given two functions $f, g : X \to \{0,1\}^n$, with the promise that there exists a unique $S \in \{0,1\}^n$ such that $\forall(x,y),\ f(x) \oplus g(y) \neq S$, recover S. We further assume that f and g behave like random functions, and that we are given a set $\mathcal{S} \subseteq \{0,1\}^n$, such that $S \in \mathcal{S}$ ($\mathcal{S}$ represents prior knowledge about the secret). In an attack against the counter mode, f outputs correspond to known keystream blocks, while g outputs correspond to encryptions of S.

In the information theoretic setting, this problem can be solved with $\tilde{\mathcal{O}}(2^{n/2})$ queries for any set $\mathcal{S}$, and requires at least $\Omega(2^{n/2})$ queries when $|\mathcal{S}| \geq 2$. However, the analysis is more complex when taking into account the cost of the computations required to recover S. McGrew introduces two algorithms for this problem: a sieving algorithm with $\tilde{\mathcal{O}}(2^{n/2})$ queries and time $\tilde{\mathcal{O}}(2^n)$, and a searching algorithm that can be optimized to time and query complexity $\tilde{\mathcal{O}}(2^{n/2}\sqrt{|\mathcal{S}|})$. Our main contribution is to give better algorithms for this problem:

1. An algorithm with $\tilde{\mathcal{O}}(2^{n/2})$ queries and time $\tilde{\mathcal{O}}(2^{n/2} + 2^{\dim\langle\mathcal{S}\rangle})$, in the case where $\mathcal{S}$ is (a subset of) a linear subspace of $\{0,1\}^n$. In particular, when $\mathcal{S}$ is a linear subspace of dimension $n/2$, we reach a time and query complexity of $\tilde{\mathcal{O}}(2^{n/2})$, while the searching algorithm of McGrew has a time and query complexity of $\tilde{\mathcal{O}}(2^{3n/4})$.
2. An algorithm with time and query complexity $\tilde{\mathcal{O}}(2^{2n/3})$ for any $\mathcal{S}$. In particular, with $\mathcal{S} = \{0,1\}^n$, the best previous algorithm had a time complexity of $\tilde{\mathcal{O}}(2^n)$.

We also show new applications of these algorithms. The first algorithm leads to an efficient message recovery attack with complexity $\tilde{\mathcal{O}}(2^{n/2})$ against the CTR mode, assuming that the adversary can control the position of the secret, by splitting it across block boundaries (following ideas of [32] and [12]). The second algorithm can be used to recover the polynomial key in some polynomial based MACs such as GMAC and Poly1305, leading to a universal forgery attack with complexity $\tilde{\mathcal{O}}(2^{2n/3})$. As far as we know, this is the first universal forgery attack against those MACs with complexity below 2^n.

Related works. There are several known results about the security of mode of operation beyond the birthday bound, when the proof is not applicable. For encryption modes, the security of the CBC mode beyond the birthday bound is well understood: collision attacks reveal the XOR of two message blocks, and can exploited in practice [7]. Other modes that allow collisions (eg. CFB) have the same properties. The goal of this paper is to study the security of modes that don't have collisions, to get a similar understanding of their security.

Many interesting attacks have also been found against authentication modes. In 1995, Preneel and van Oorschot [31] gave a generic collision attack against all deterministic iterated message authentication codes (MACs), leading to existential forgeries with complexity $\mathcal{O}(2^{n/2})$. Later, a number of more advanced generic attacks have been described, with stronger outcomes than existential forgeries, starting with a key-recovery attack against the envelop MAC by the same authors [32]. In particular, a series of attack against hash-based MAC [25,30,18,11] led to universal forgery attacks against long challenges, and key-recovery attacks when the hash function has an internal checksum (like the GOST family). Against PMAC, Lee *et al.* showed a universal forgery attack in 2006 [24]. Later, Fuhr, Leurent and Suder gave a key-recovery attack against the PMAC variant used in AEZv3 [17]. Issues with GCM authentication with truncated tags were also pointed out by Ferguson [14].

None of these attacks contradict the proof of security of the scheme they target, but they are important results to understand the security degradation after the birthday bound.

Organization of the paper. We introduce the CTR mode and the missing difference problem in Section 2, and present our algorithmic contributions in Section 3. Then we describe concrete attacks against the CTR mode in Section 4, and attacks against Carter-Wegman MACs in Section 5. At last we show detailed proofs and simulation results in Section 6.

2 Message Recovery Attacks on CTR mode

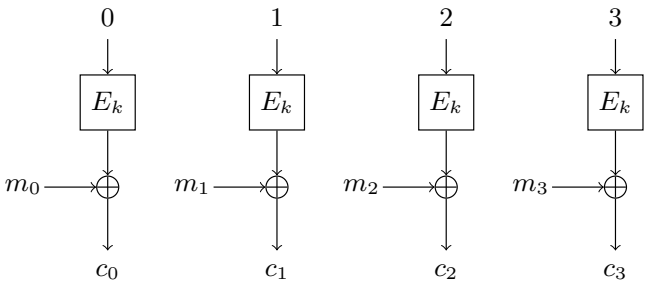


Fig. 1. CTR mode

The CTR mode was first proposed by Diffie and Hellman in 1979 [10]. It was not included in the first series of standardized modes by NIST [16], but was added later [13]. The CTR mode essentially turns a block cipher into a stream cipher, by encrypting some non-repeating counter. It is now a popular mode of operation, thanks to its parallelizability, speed, and simple design. This led Phillip Rogaway to write in an evaluation of different privacy modes of operation talking about CTR [35]: "Overall, usually the best and most modern way to achieve privacy-only encryption". In particular, CTR is used as the basic of the authenticated encryption mode GCM, the most widely used mode in TLS today.

2.1 Setting and Notations

In the following we assume that the counter mode is implemented such that the input to the block cipher never repeats. For simplicity we consider a stateful variant of the counter mode with a global counter that is maintained across messages and initialized as 0 (as shown in Figure 1):

$$c_i = E_k(i) \oplus m_i,$$

where E_k is an n-bit block cipher, m_i an n-bit block of plaintext and c_i an n-bit block of ciphertext.

Our attacks do not depend on the details of how the input to the block cipher is constructed, and can also be applied to nonce-based variants[1]; we only require that all inputs are different. Note that some variants of the counter mode can have repetitions in the block cipher input[2], but this gives easy attacks because repetitions leak the xor of two plaintext blocks (as in the CBC mode).

We consider a message recovery attack, where the attacker tries to recover some secret message block S. Throughout the attack, the key k will be invariant so we will write $E_k(i)$ as a_i to represent the i^{th} block of CTR keystream. We can immediately notice that if we have partial knowledge of the plaintext, for every known block m_i we can recover the associated a_i as $c_i \oplus m_i = a_i$. Assume further that we have access to the repeated encryption b_j of the secret S so that $b_j = a_j \oplus S$. The first property of the CTR mode is that $E_k(\cdot)$ being a permutation, the keystream a_i never repeats, thus we have the following inequalities:

$$i \neq j \Rightarrow a_i \neq a_j \Rightarrow a_i \oplus a_j \oplus S \neq S \Rightarrow a_i \oplus b_j \neq S.$$

From now on we will always assume that we can observe and collect lists of many a_i and b_j and use them with the previous inequality to recover S. This setting is similar to the practical attack Sweet32 on the CBC mode mounted by Bhargavan and Leurent, using repeated encryptions of an authentication token to obtain many different ciphertext blocks for the same secret information [7].

Formally, let $\mathcal{A} \subseteq \{0,1\}^n$ be the set of observed keystream blocks, $\mathcal{B} \subseteq \{0,1\}^n$ the set of observed encryptions and $\mathcal{S} \subseteq \{0,1\}^n$ the set of possible secrets

[1] For instance, GCM concatenates a per-message nonce and a counter within a message.

[2] For instance, the treatment of non-default-length nonces in GCM can lead to collisions [23].

(corresponding to some already known information about S). We define the missing difference algorithmic problem in terms of set:

Definition 1 (Missing Difference Problem). *Given two sets $\mathcal{A}$ and $\mathcal{B}$, and a hint $\mathcal{S}$, find the value $S \in \mathcal{S}$ such that:*

$$\forall (a, b) \in \mathcal{A} \times \mathcal{B},\ S \neq a \oplus b\ .$$

Alternatively, we can consider that the attacker is given oracle access to $\mathcal{A}$ and $\mathcal{B}$ though some functions f and g, so that its running time includes calls to f and g, and computations to recover S. This presentation corresponds to a more active attack, where the adversary can optimize the size of the sets.

Definition 2 (Missing Difference Problem with Functions). *Given two functions $f, g : X \to \{0,1\}^n$, and a hint $\mathcal{S}$, find the value $S \in \mathcal{S}$ such that:*

$$\forall (x, y),\ S \neq f(x) \oplus g(y)\ .$$

2.2 Previous Work

An attack can only be carried to the end if the secret S is the only value in $\mathcal{S}$ such that $\forall (a, b) \in \mathcal{A} \times \mathcal{B},\ S \neq a \oplus b$, or else it will be indistinguishable from the other values that satisfy the same condition (those values could have produced the same sets with same probability). The coupon collector's problem predicts that N out of N different coupons are found after $N \cdot H_N \simeq N \ln N$ draws (with H_N the N-th harmonic number), assuming uniform distribution of the draws. In our case we will assume that all the differences $a \oplus b$ are independent and uniformly distributed over $\{0,1\}^n \setminus S$, which is a reasonable approximation validated by our experiments. To carry the attack to the end we require to collect $N = |\mathcal{S}| - 1$ differences thus we will need $\mathcal{O}(|\mathcal{S}| \ln |\mathcal{S}|)$ "draws". A draw is a couple (a, b) s.t. $a \oplus b \in |\mathcal{S}|$, otherwise we discard it; it happens with probability $(|\mathcal{S}| - 1)/(2^n - 1)$. Therefore we need to observe enough data to have $|\mathcal{A}| \cdot |\mathcal{B}|$ in the order of $\mathcal{O}(2^n \ln |\mathcal{S}|)$; this may be achieved by having both sets in the order of $\mathcal{O}(2^{n/2}\sqrt{\ln |\mathcal{S}|})$. This size of the observed sets can be understood as the query complexity, that is the number of encrypted messages the attacker will have to intercept in order to carry out the attack. Notice that even for $|\mathcal{S}| = \mathcal{O}(2^n)$, $|\mathcal{A}| = |\mathcal{B}| = \mathcal{O}(\sqrt{n} \cdot 2^{n/2})$ is quite close to the theoretical lower bound of $\mathcal{O}(2^{n/2})$ given by the distinguishing attack and the security proof for the CTR mode. Therefore, message recovery attacks are possible with an (almost) optimal data complexity. The next question is to study the time complexity, *i.e.* how to efficiently recover S.

A first approach consists in computing all the impossible values of S from the large set of $\mathcal{A} \times \mathcal{B}$ and discard any new value we encounter as impossible until there's only one possible plaintext left. This is algorithm 1. This approach works but requires to actually compute $\mathcal{O}(2^n \ln |\mathcal{S}|)$ values and maintain in memory a sieve of size $|\mathcal{S}|$. In the case where the key size is equal to the block size n, like AES-128, this attack is actually worse than a simple exhaustive search of the key.

Algorithm 1. Simple sieving algorithm

Input: $\mathcal{A}, \mathcal{B}, \mathcal{S}$
Output: $\{s \in \mathcal{S} \mid \forall (a,b) \in \mathcal{A} \times \mathcal{B},\ a \oplus b \neq s\}$
  for a **in** $\mathcal{A}$ **do**
    for b **in** $\mathcal{B}$ **do**
      Remove $(a \oplus b)$ from $\mathcal{S}$;
    end for
  end for
  return $\mathcal{S}$

Algorithm 2. Searching algorithm

Input: $\mathcal{A}, \mathcal{B}, \mathcal{S}$
Output: $\{s \in \mathcal{S} \mid \forall (a,b) \in \mathcal{A} \times \mathcal{B},\ a \oplus b \neq s\}$
  Store $\mathcal{B}$ so that operation $\in$ is efficient.
  for s **in** $\mathcal{S}$ **do**
    for a **in** $\mathcal{A}$ **do**
      if $(s \oplus a) \in \mathcal{B}$ **then**
        Remove s from $\mathcal{S}$;
      end if
    end for
  end for
  return $\mathcal{S}$

In a 2012 work, McGrew [28] described this sieving algorithm and noticed that when the set $\mathcal{S}$ is small, the sieving wastes a lot of time computing useless values. Therefore he proposed a second algorithm, algorithm 2, to test and eliminate values of $\mathcal{S}$ one by one. This algorithm loops over $\mathcal{S}$ and $\mathcal{A}$ to efficiently test whether $s \oplus a \in \mathcal{B}$; if yes then we sieve the value s out of $\mathcal{S}$.

Both algorithms act on a sieving set $\mathcal{S}$ to reduce it, so McGrew proposed a hybrid algorithm switching from one algorithm to the other in order to reduce the searching space as quickly as possible. This improves the attack when $\mathcal{A}$ and $\mathcal{B}$ are fixed, but if the adversary can choose the sizes of $\mathcal{A}$ and $\mathcal{B}$ (in particular, if he actually has oracle access to functions f and g), then the searching algorithm allows better trade-offs. Indeed, the searching algorithm has a complexity of $\mathcal{O}(|\mathcal{B}|+|\mathcal{A}|\cdot|\mathcal{S}|)$, and is successful as soon as $|\mathcal{A}|\cdot|\mathcal{B}| = \Omega(2^n \ln |\mathcal{S}|)$. To optimize the complexity, we use $|\mathcal{B}| = |\mathcal{A}| \cdot |\mathcal{S}|$ to obtain an overall complexity of $\mathcal{O}(2^{n/2}\sqrt{|\mathcal{S}| \ln |\mathcal{S}|})$ in both time and queries. In particular for small $\mathcal{S}$ (of size polynomial in n) this algorithm is (almost) optimal, reaching the birthday bound $\tilde{\mathcal{O}}(2^{n/2})$.

Starting from these observations we will show improved algorithms to recover a block of secret information without big exhaustive searches in the next section.

3 Efficient Algorithms for the Missing Difference Problem

We now propose two new algorithms to solve the missing difference algorithmic problem more efficiently in two practically relevant different settings. Our first algorithm requires that the set $\mathcal{S}$ — or its linear span $\langle\mathcal{S}\rangle$ — is a vector space of relatively small dimension, and has complexity $\tilde{\mathcal{O}}(2^{n/2} + |\langle\mathcal{S}\rangle|)$. The second algorithm uses a larger query complexity of $\tilde{\mathcal{O}}(2^{2n/3})$, to reduce the computation and memory usage to $\tilde{\mathcal{O}}(2^{2n/3})$.

3.1 Known Prefix Sieving

In many concrete attack scenarios, an attacker knows some bits of the secret message in advance. For instance, an HTTP cookie typically uses ASCII printable characters, whose high order bit is always set to zero. More generally, we assume that $\mathcal{S}$ is (included in) an affine subspace of $\{0,1\}^n$ of dimension $n-z$ for some natural $z < n$. In order to simplify the attack, we use a bijective affine function ϕ that maps $\mathcal{S}$ unto $\{0\}^z \times \{0,1\}^{n-z}$, and rewrite the problem as follows:

$$\begin{aligned} S \neq a \oplus b &\Leftrightarrow \phi(S) \neq \phi(a \oplus b), && \text{as } \phi \text{ is a bijection.} \\ &\Leftrightarrow \phi(S) \neq \phi(a) \oplus \phi(b) \oplus \phi(0), && \text{as } \phi \text{ is affine} \end{aligned}$$

Therefore, we can reduce the missing difference problem on $\mathcal{A}$, $\mathcal{B}$, $\mathcal{S}$ with $\dim(\langle\mathcal{S}\rangle) = n - z$ to the missing difference problem on $\mathcal{A}'$, $\mathcal{B}'$, $\mathcal{S}'$, where the secret is known to start with z zeroes:

$$\begin{aligned} \mathcal{S}' &:= \{0\}^z \times \{0,1\}^{n-z} \\ \mathcal{A}' &:= \{\phi(a) \mid a \in \mathcal{A}\} \\ \mathcal{B}' &:= \{\phi(b) \oplus \phi(0) \mid b \in \mathcal{B}\} \end{aligned}$$

We now introduce a known prefix sieving algorithm (Algorithm 3) to solve this problem efficiently. The algorithm is quite straightforward; it looks for a prefix collision before sieving in the same way as before to recover S. The complexity depend on the dimension $n - z$; the sieving requires $\mathcal{O}(2^{n-z})$ memory and $\mathcal{O}((n-z) \cdot 2^{n-z})$ XOR computations in expectation, while looking for collisions only requires to store the prefix keys and to go through one of the set. Looking for collisions allows us to skip the computations of many pairs (a, b) that would be irrelevant as $a \oplus b \notin \mathcal{S}$.

The expected number of collisions required to isolate the secret is given by the coupon collector problem as $\ln(2^{n-z})2^{n-z} = \ln 2 \cdot (n-z) \cdot 2^{n-z}$. Therefore the total optimized complexity (with balanced sets $\mathcal{A}$ and $\mathcal{B}$) to recover an $n - z$ bits secret with this algorithm is:

$$\begin{aligned} &\mathcal{O}\left(\sqrt{n-z} \cdot 2^{n/2}\right) && \text{queries} \\ &\mathcal{O}\left(2^{n-z} + n\sqrt{n-z} \cdot 2^{n/2}\right) && \text{bits of memory (sieving \& queries)} \\ &\mathcal{O}\left((n-z) \cdot 2^{n-z} + \sqrt{n-z} \cdot 2^{n/2}\right) && \text{operations (sieving \& collisions searching)} \end{aligned}$$

As we can see from the complexity, when $z = 0$ this is the naive algorithm with its original complexity. When z nears n, this performs similarly to McGrew's searching algorithm *i.e.* the cost of looking for collisions (or storing $\mathcal{B}$ so that the search is efficient) will dominate the overall cost of the algorithm therefore the time and query complexity will match. Actually, this algorithm improves over previous works for intermediate values of z. With $z = n/2$, we have an algorithm with complexity $\tilde{\mathcal{O}}(2^{n/2})$, while McGrew's searching algorithm would require $\tilde{\mathcal{O}}(2^{3n/4})$ computations in the same setting. The complexity therefore becomes tractable and we could implement and run this algorithm for $n = 64$ bits with success, as shown in Section 6.2.

Algorithm 3. Known prefix sieving algorithm

Input: n, $z < n$, $\mathcal{A}, \mathcal{B}$, $\mathcal{S} \subseteq \{0\}^z \times \{0,1\}^{n-z}$
Output: $\{s \in \mathcal{S} \mid \forall (a,b) \in \mathcal{A} \times \mathcal{B},\ a \oplus b \neq s\}$

```
h_B ← Empty hash table.
for b in B do
    h_B[b_[0...(z-1)]] ∪← {b_[z...(n-1)]}
end for

for a in A do
    v_a ← a_[z...(n-1)]
    for v_b in h_B[a_[0...(z-1)]] do
        Remove 0 || (v_a ⊕ v_b) from S;
    end for
end for
return S
```

3.2 Fast Convolution Sieving

Alternatively, we can reduce the complexity of the sieving algorithm by using sets $\mathcal{A}$ and $\mathcal{B}$ of size $2^m \gg 2^{n/2}$, rather than $\tilde{\mathcal{O}}(2^{n/2})$ as required to uniquely identify S. If we consider all the values $a \oplus b$ for (a, b) in $\mathcal{A} \times \mathcal{B}$, we expect that they are close to uniformly distributed over $\{0,1\}^n \setminus S$, so that every value except S is reached about 2^{2m-n} times, while S is never hit. Increasing m makes the gap more visible than with sets of size only $\tilde{\mathcal{O}}(2^{n/2})$. Therefore, we can consider buckets of several candidates s, and accumulate the number of $a \oplus b$ in each bucket. If we consider buckets of 2^t values, each bucket receives 2^{2m+t-n} values on average, but the bucket containing S receives only $2^{2m+t-n} - 2^{2m-n}$ values. If we model this number with random variables following a binomial distribution, the variance σ^2 is about $2^{m+t/2-n/2}$. Therefore, the bias will be detectable when: $\sigma \ll 2^{2m-n}$, *i.e.* when $t \ll 2m - n$.

Concretely, we use a truncation function T that keeps only $n - t$ bits of an n-bit word. We consider the values $T(a \oplus b)$ for all $(a, b) \in \mathcal{A} \times \mathcal{B}$, and count how

many times each value is reached. If m is large enough, the value with the lowest counter corresponds to $T(S)$. This attack does not require any prior information on the secret; it can be used with $\mathcal{S} = \{0,1\}^n$, and once $T(S)$ is known, we can use known-prefix sieving to recover the remaining bits (looking for S in an affine space of dimension t).

We now show an algorithm to quickly count the number of occurrences for each combination. For a given multi-set $\mathcal{X}$, we consider an array of counters $C_{\mathcal{X}}$, to represent how many times each value $T(x)$ is reached:

$$C_{\mathcal{X}}[i] = \left|\left\{x \in \mathcal{X} \mid T(x) = i\right\}\right|.$$

Our goal is to compute $C_{\mathcal{A}\oplus\mathcal{B}}$ efficiently from $\mathcal{A}$ and $\mathcal{B}$, where $\mathcal{A} \oplus \mathcal{B}$ is the multi-set $\{a \oplus b \mid (a,b) \in \mathcal{A} \times \mathcal{B}\}$. We observe that:

$$\begin{aligned}
C_{\mathcal{A}\oplus\mathcal{B}}[i] &= |\{(a,b) \in \mathcal{A} \times \mathcal{B} \mid T(a \oplus b) = i\}| \\
&= \sum_{a \in \mathcal{A}} |\{b \in \mathcal{B} \mid T(a \oplus b) = i\}| \\
&= \sum_{a \in \mathcal{A}} |\{b \in \mathcal{B} \mid T(b) = i \oplus T(a)\}| \\
&= \sum_{a \in \mathcal{A}} C_{\mathcal{B}}[i \oplus T(a)] \\
&= \sum_{j \in \{0,1\}^{n-t}} C_{\mathcal{A}}[j] C_{\mathcal{B}}[i \oplus j]
\end{aligned}$$

This is a form of convolution that can be computed efficiently only using the Fast Walsh-Hadamard Transform (Algorithm 4), in the same way we use the Fast Fourier Transform to compute circular convolutions (see Algorithm 5). Therefore the full attack (shown in Algorithm 6) takes time $\tilde{\mathcal{O}}(2^{n-t})$ using lists of size 2^m with $m \gg (n+t)/2$ and a sieve of 2^{n-t} elements.

In order to optimize the attack, we select $t = n/3$ such that the time complexity, data complexity, and memory usage are all roughly $2^{2n/3}$. A detailed analysis in Section 6.1 shows that we reach a constant success rate with $t = n/3$ using lists of size $\mathcal{O}(\sqrt{n} \cdot 2^{2n/3})$. This gives the following complexity for the full attack:

$$\begin{aligned}
&\mathcal{O}(\sqrt{n} \cdot 2^{2n/3}) && \text{queries} \\
&\mathcal{O}(n \cdot 2^{2n/3}) + \mathcal{O}(n\sqrt{n} \cdot 2^{n/2}) && \text{bits of memory (counters + sieving)} \\
&\mathcal{O}(n \cdot 2^{2n/3}) + \mathcal{O}(n\sqrt{n} \cdot 2^{n/2}) && \text{computations (fast Walsh-Hadamard + sieving)}
\end{aligned}$$

As seen in Section 6.1, we performed experiments with $n = 12, 24, 48$, and the correct S was found with the lowest counter in at least 70% of our experiments, using list of size $\sqrt{n}2^{2n/3}$. This validates our approach and shows that the constant in the $\mathcal{O}$ notation is small. We could run this algorithm over $n = 48$ bits in a matter of minutes.

Optimizations. In order to increase the success rate of the algorithm, one can test several candidates for $T(S)$ (using the lowest remaining counters), and use

Algorithm 4. Fast Walsh-Hadamard Transform

Input: $C_{\mathcal{A}},\ |C_{\mathcal{A}}| = 2^m$
Output: The Walsh-Hadamard transform of $C_{\mathcal{A}}$
  for $d = m$ **downto** 0 **do**
    for $i = 0$ **to** 2^{m-d} **do**
      for $j = 0$ **to** 2^{d-1} **do**
        $C_{\mathcal{A}}[i \cdot 2^d + j] \leftarrow C_{\mathcal{A}}[i \cdot 2^d + j] + C_{\mathcal{A}}[i \cdot 2^d + j + 2^{d-1}]$
        $C_{\mathcal{A}}[i \cdot 2^d + j + 2^{d-1}] \leftarrow C_{\mathcal{A}}[i \cdot 2^d + j] - 2 \cdot C_{\mathcal{A}}[i \cdot 2^d + j + 2^{d-1}]$
      end for
    end for
  end for
  return $C_{\mathcal{A}}$

Algorithm 5. Fast convolution

Input: $C_{\mathcal{A}}, C_{\mathcal{B}}$
Output: $C_{\mathcal{A}\oplus\mathcal{B}}$
  {Perform fast Walsh-Hadamard transform in-place}
  FWHT($C_{\mathcal{A}}$); FWHT($C_{\mathcal{B}}$);
  for $c = 0$ **to** 2^{n-t} **do**
    $C_{\mathcal{A}\oplus\mathcal{B}}[c] \leftarrow C_{\mathcal{A}}[c] \cdot C_{\mathcal{B}}[c]$
  end for
  {Perform fast Walsh-Hadamard transform in-place}
  FWHT($C_{\mathcal{A}\oplus\mathcal{B}}$);
  return $C_{\mathcal{A}\oplus\mathcal{B}}$

Algorithm 6. Sieving with fast convolution

Input: $\mathcal{A}, \mathcal{B}, t \leq n$
Output: S s.t. $\forall (a,b) \in \mathcal{A} \times \mathcal{B},\ a \oplus b \neq S$
  $C_{\mathcal{A}}, C_{\mathcal{B}}, C_{\mathcal{A}\oplus\mathcal{B}} \leftarrow$ arrays of 2^{n-t} integers initialized to 0;
  for a **in** $\mathcal{A}$ **do**
    Increment $C_{\mathcal{A}}[a_{0..(n-t-1)}]$
  end for
  for b **in** $\mathcal{B}$ **do**
    Increment $C_{\mathcal{B}}[b_{0..(n-t-1)}]$
  end for
  $C_{\mathcal{A}\oplus\mathcal{B}} \leftarrow$ FastConvolution($C_{\mathcal{A}}, C_{\mathcal{B}}$)
  $u \leftarrow \mathrm{argmin}_i\, C_{\mathcal{A}\oplus\mathcal{B}}[i]$
  Run known prefix sieving (Algorithm 3), knowing that $T(S) = u$

the known-prefix sieving to detect whether the candidate is correct. Another option is to run multiple independent runs of the algorithm with different choices of the $n/3$ truncated bits. This would avoid some bad cases we have observed in simulations, where the right counter grows abnormally high and gets hidden in all of the other counters.

For the memory complexity, notice that we don't need to store all the data but simply to increment a counter. We only need to keep enough blocks for the second part of the algorithm so that the sieving yields a unique result. Initially the counters for $C_{\mathcal{A}}$ and $C_{\mathcal{B}}$ are quite small, $\sqrt{n}$ in expectation. However, $C_{\mathcal{A}\oplus\mathcal{B}}$ will have much bigger entries, $n \cdot 2^{2n/3}$ in expectation, so that we need $\mathcal{O}(n)$ bits to store each entry.

4 Application to the CTR Mode

We know show how to mount attacks against the counter mode using the new algorithms for the missing difference problem.

4.1 Attack using Fast Convolution

Use of the fast convolution algorithm to recover one block of CTR mode plaintext is straightforward. The attacker is completely passive and observes encryptions of S (gathered in set $\mathcal{B}$), and keystream blocks recovered from the encryption of known message blocks (gathered in set $\mathcal{A}$). When the lists are large enough, he runs the fast convolution algorithm on $\mathcal{A}$ and $\mathcal{B}$ to recover S.

4.2 Attacks using Known Prefix Sieving

Direct attack. There are many settings where unknown plaintext will naturally lie in some known affine subspace, and the known prefix sieving algorithm can be used directly. For instance a credit card number (or any number) could be encoded in 16 bytes of ASCII then encrypted. Because in ASCII the encoding of any digit starts by `0x3` (`0x30` to `0x39`), we know half of the bits of the plaintext, and we can use the known-prefix sieving with $z = n/2$. Other examples are information encoded by `uuencode` that uses ASCII values `0x20` to `0x5F` (corresponding to two known bits) or HTML authentication cookies that are typically encoded to some subset of ASCII numbers and letters[3].

Block splitting. We often assume that the secret is encrypted in its own block, but when the secret is part of the message, it can also be split across block boundaries, depending on how the plaintext is constructed and encrypted by the protocol. In particular, if a message block contains both known bytes and secret

[3] For example, `wikipedia.org` encodes cookies with lower case letters and digits, this corresponds to two known bits.

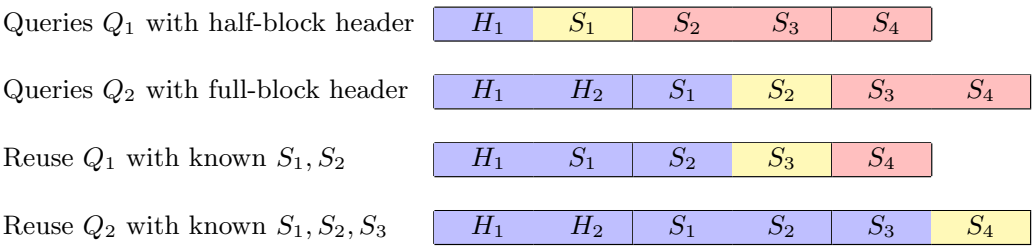


Table 1. Example of an attack on two blocks secret $S = S_1 \parallel S_2 \parallel S_3 \parallel S_4$. Each step performs the known prefix sieving algorithm. Known information in blue, unknown information in red, attacked information in yellow.

bytes, we can apply the known prefix sieving algorithm to this block and recover the secret bytes.

In many protocols, messages start with some low entropy header that can be guessed by an attacker. Moreover, the attacker often has some degree of control over those headers. For instance, in the BEAST attack [12] against HTTPS, an attacker uses Javascript code to generate HTTPS requests, and he can choose the URL corresponding to the requests. Using this control of the length of the header, block splitting attacks have been shown in the BEAST model [12,20]. The attacker starts with a header length so that a small chunk of the secret message is encrypted together with known information, and recovers this secret chunk. Then he changes the length of the header to recover a second chunk of the message, using the fact that the first chunk is now known. Eventually, the full secret can be recovered iteratively.

In our case, the easiest choice is to recover chunks of $n/2$ bits of secret one by one, using the known-prefix sieving algorithm with $z = n/2$. We illustrate this attack in Table 1, assuming a two-block secret $S = S_1 \parallel S_2 \parallel S_3 \parallel S_4$, and a protocol that lets the adversary query an encryption of the secret with an arbitrary chosen prefix:

1. The attacker makes two kind of queries
   - Q_1 with a known half-block header H_1 ($\mathcal{E}([H_1 \parallel S_1] \parallel [S_2 \parallel S_3] \parallel [S_4])$);
   - Q_2 with a known full-block header $H_1 \| H_2$ ($\mathcal{E}([H_1 \| H_2] \| [S_1 \| S_2] \| [S_3 \| S_4])$).
2. He first recovers S_1 using the known-prefix sieving with the first block of each type of query. More precisely, he uses $\mathcal{A} = \{\mathcal{E}(H_1 \parallel H_2)\}$ and $\mathcal{B} = \{\mathcal{E}(H_1 \parallel S_1)\}$, so that the missing difference is $0 \parallel (S_1 \oplus H_2)$.
3. When S_1 is known, he can again use known prefix sieving to recover S_2, with the first and second blocks of Q_2 queries: $\mathcal{A} = \{\mathcal{E}(H_1 \parallel H_2)\}$ and $\mathcal{B} = \{\mathcal{E}(S_1 \parallel S_2)\}$, so that the missing difference is $(S_1 \oplus H_1) \parallel (S_2 \oplus H_2)$. To improve the success rate of this step, he can also consider the first block of Q_1 queries as known keystream.
4. When S_2 is known, another round of known prefix sieving reveals S_3, *e.g.* with $\mathcal{A} = \{\mathcal{E}(H_1 \parallel H_2)\}$ and $\mathcal{B} = \{\mathcal{E}(S_2 \parallel S_3)\}$, the missing difference is $(S_2 \oplus H_1) \parallel (S_3 \oplus H_2)$.

5. Finally, S_4 is recovered with a last round of known prefix sieving using $\mathcal{A} = \{\mathcal{E}(H_1 \parallel H_2)\}$ and $\mathcal{B} = \{\mathcal{E}(S_3 \parallel S_4)\}$, with missing difference is $(S_3 \oplus H_1) \parallel (S_4 \oplus H_2)$.

This gives an algorithm with query complexity of $\mathcal{O}(\sqrt{n}2^{n/2})$ to recover repeated encryption of a secret over multiple blocks in the BEAST attacker model. In Section 6.2, we analyze the constants in the $\mathcal{O}()$ and run experiments with $n = 64$ using locally encrypted data. In particular, we have a success probability higher than 80% using two lists of 5×2^{32} queries with $n = 64$.

More generally, we show that for $n \geq 32$ the success probability of this attack is at least 99% with lists of size $\sqrt{n/2} \cdot 2^{n/2}$. With a one block secret, an optimal attack uses two lists of $\sqrt{n/2} \cdot 2^{n/2}$ two-block queries: queries $[H_1 \parallel S_1] \parallel [S_2]$ with a half-block header, and queries $[H_1 \parallel H_2] \parallel [S_1 \parallel S_2]$ with a full-block header. This translates to a data complexity of $4\sqrt{n/2} \cdot 2^{n/2}$ blocks. For comparison, an attack against the CBC mode requires on average $2 \cdot 2^{n/2}$ blocks of data in the ideal case.

Alternatively, an attacker could recover the secret bit by bit. This leads to a more complex attack in practice, but the complexity is similar, and this variant could use McGrew's searching algorithm instead of our known-prefix sieving algorithm (because in this scenario, we have $|\mathcal{S}| = 2$). We show a detailed analysis of this variant in Section 6.2, taking into account the n steps necessary for this attack.

4.3 Use of CTR Mode in Communication Protocols

The CTR mode is widely used in internet protocols, in particular as part of the GCM authenticated encryption mode [29], with the AES block cipher. For instance, Mozilla telemetry data show that more than 90% of HTTPS connections from Firefox 58 use AES-GCM[4]. While attacks against modes with a 128-bit block cipher are not practical yet, it is important to limit the amount of data processed with a given key, in order to keep the probability of a successfull attack negligible, following the guidelines of Luykx and Paterson [26].

Surprisingly, there are also real protocols that use 64-bit block ciphers with the CTR mode (or variants of the CTR mode), as shown below. Attacks against those protocols would be (close to) practical, assuming a scenario where an attacker can generate the encryption of a large number of messages with some fixed secret.

SSH. Ciphersuites based on the CTR mode were added to SSHv2 in 2006 [4]. In particular, 3DES-CTR is one of the recommended ciphers, but actual usage of 3DES-CTR seems to be rather low [1]. In practice, 3DES-CTR is optionally supported by the dropbear server, but it is not implemented in OpenSSH. According to a scan of the full IPv4 space by Censys.io[5], around 9% of SSH servers

[4] `https://mzl.la/2GY53Mc`, accessed February 8, 2018

[5] `https://censys.io/data/22-ssh-banner-full_ipv4`, scan performed July 5, 2017

support 3DES-CTR, but actual usage is hard to estimate because it depends on client configuration.

The SSH specification requires to rekey after 1GB of data, but an attack is still possible, although the complexity increases.

3G telephony. The main encryption algorithm in UMTS telephony is based on the 64-bit blockcipher Kasumi. The mode of operation, denoted as f8, is represented in Figure 2. While this mode in not the CTR mode and was designed to avoid its weaknesses, our attack can be applied to the first block of ciphertext. Indeed the first block of message i is encrypted as $c_{i,0} = m_{i,0} \oplus E_k(E_{k'}(i))$, where the value $E_k(E_{k'}(i))$ is unique for all the messages encrypted with a given key.

There is a maximum of 2^{32} messages encrypted with a given key in 3G, but this only has a small effect on the complexity of attacks.

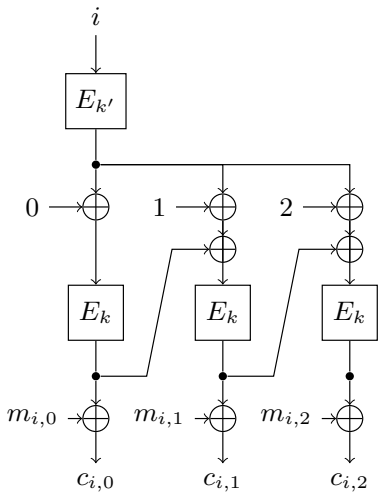


Fig. 2. f8 mode (i is a message counter)

Because of the low usage of 3DES-CTR in SSH, and the difficulty of mounting an attack against 3G telephony in practice, we did not attempt to demonstrate the attack in practice, but the setting and complexity of our attacks are comparable to recent results on the CBC mode with 64-bit ciphers [7].

4.4 Counter-measures.

As for many modes of operation, the common wisdom to counter this kind of attacks asks for rekeying before the birthday bound, *i.e.* before $2^{n/2}$ blocks. However rekeying too close to the birthday bound may not be enough. For example let's consider an implementation of a CTR based mode of operation that rekeys every $2^{n/2}$ blocks, Using the same model as previously, and a one-block

secret, an optimal attack uses queries $[H_1 \parallel S_1] \parallel [S_2]$ with a half-block header, and queries $[H_1 \parallel H_2] \parallel [S_1 \parallel S_2]$ with a full-block header, where rekeying occurs after $2^{n/2-2}$ queries of each type. To recover S_1, we use the known prefix sieving algorithm as previously, but we can only use relations between ciphertext blocks encrypted with the same key. In each session of $2^{n/2}$ blocks, we consider 2^{n-4} pairs of ciphertext blocks; on average there are $2^{n/2-4}$ pairs with the correct prefix used for sieving. Since we need $n/2 \cdot 2^{n/2}$ draws to reduce the sieve to a single element with high probability, we use $8n$ sessions, *i.e.* $8n \cdot 2^{n/2}$ blocks of data in total. The same data can be reused to recover S_2 when S_1 is known. This should be compared with the previous data complexity of $4\sqrt{n/2} \cdot 2^{n/2}$ in the absence of rekeying.

However, rekeying every $2^{n/2-16}$ blocks makes the data complexity goes up to $2^{35}n$ sessions or $n \cdot 2^{19+n/2}$ blocks to recover the secret block. Notice that the security gain of rekeying is comparable with what is gained in CBC, where rekeying every $2^{n/2-16}$ blocks forces increases the data complexity from $2 \cdot 2^{n/2}$ to $2^{18} \cdot 2^{n/2}$.

5 Application to Wegman-Carter MACs

Because the fast convolution algorithm requires fewer assumptions, it can be adapted to other modes of operation based on CTR and particularly to Wegman-Carter type of constructions for MAC. Wegman-Carter MACs use a keyed permutation E and a keyed universal hash function h, with $k1$ and $k2$ two private keys. The input is a message M and a nonce N, and the MAC is defined as:

$$\mathrm{MAC}(N, M) = h_{k1}(M) + E_{k2}(N)$$

Again, the construction requires that all block cipher inputs are different. To apply our attack, we use two fixed message M and M', and we capture many values $\mathrm{MAC}(N, M)$ in a list $\mathcal{A}$ and values $\mathrm{MAC}(N', M')$ in a list $\mathcal{B}$, all using unique nonces. Then we solve the missing difference problem to recover $h_{k1}(M) - h_{k1}(M')$ as we know that $\forall N \neq N' : E_{k2}(N) - E_{k2}(N') \neq 0$. It is often sufficient to know this difference and the two messages M and M' to recover the key k_1. We give two examples with concrete MAC algorithms.

Galois/Counter Mode. GCM is an authenticated encryption mode with associated data, combining the CTR mode for encryption and a Wegman-Carter MAC based on polynomial evaluation in a Galois field for authentication. It takes as input a message M that is encrypted and authenticated, and some associated data A that is authenticated but not encrypted. When used with an empty message, the resulting MAC is known as GMAC. In our attack, we use an empty message with one block of authenticated data A, so that the tag is computed as:

$$\mathrm{MAC}(N, A) = A \cdot H^2 \oplus H \oplus E_k(N),$$

with H the hash key and $(\cdot)$ the multiplication in a Galois Field defined by a public polynomial. So, for two different blocks of authenticated data A and A'

we collect $\mathcal{O}(\sqrt{n} \cdot 2^{2n/3})$ MACs and perform the fast convolution algorithm to recover $A \cdot H^2 \oplus H \oplus A' \cdot H^2 \oplus H = (A \oplus A') \cdot H^2$. We known $A \oplus A'$ and the field is known so we invert that value and recover H^2 then compute the square root and recover the hash key H.

Comparison with previous attacks against GMAC. There are several known attacks against GCM and GMAC, but none of them seems to allow universal forgery with just $2^{2n/3}$ blocks of data and $2^{2n/3}$ computations. In particular, Handschuh and Preneel [19] gave a weak-key attack, that can also be used to recover the hash key without weak key assumptions, using roughly $2^{n/2}$ messages of $2^{n/2}$ blocks. Later work extended these weak key properties [36,33] but an attack still requires about 2^n blocks in total when no assumptions are made about the key. We also note that these attacks require access to a verification oracle, while our attack only uses a MAC oracle.

Some earlier attacks use specific options of the GCM specifications to reach a lower complexity, but cannot be applied with standard-length IV, and tag: Ferguson [14] showed an attack when the tag is truncated, and Joux [23] gave an attack based on non-default IV lengths.

Poly1305. Poly1305 [6] is a MAC scheme following the Wegman-Carter construction, using polynomial evaluation modulo the prime number $2^{130} - 5$. It uses a keyed 128-bit permutation (usually AES), and the hash function key, r, has 106 free bits (22 bits of the key are set to 0, including in particular the 4 most significant ones). The message blocks are first padded to 129-bit values c_i. Then the MAC of a q-block message M with nonce N is defined as:

$$T(M, N) = (((c_1 r^q + c_2 r^{q-1} + ... + c_q r) \mod 2^{130} - 5) + E_k(N)) \mod 2^{128}.$$

With the same strategy as above, using two different messages M and M' we recover the missing difference

$$(((c_1 - c'_1) r^q + (c_2 - c'_2) r^{q-1} + ... + (c_q - c'_q) r) \mod 2^{130} - 5) \mod 2^{128}.$$

Moreover, we chose M and M' such that $c_i - c'_i = 0$ and $c_q - c'_q = 1$; since by design, $r < 2^{124}$ the value recovered is simply the hash key r.

Notice that Poly1305 doesn't use the XOR operation but a modular addition, and we have to adapt our algorithms to this case. Luckily, the fast convolution algorithm can easily be tweaked. First, we keep the $2n/3$ least significant bits to avoid issues the carry, something the XOR operation doesn't have. Then, when the lists of counters are up, we need to compute their cyclic convolution, which is done with a fast convolution algorithm based on the fast Fourier transform (instead of fast Walsh-Hadamard). Then we verify the value suggested by the lowest counter by running the known prefix algorithm looking for collisions on the least significant bits and sieving the modular subtraction of the most significant bits. This adaptation has similar complexities and proofs than the one described earlier. Moreover, in the case of Poly1305, one can further adapt the algorithms

to take into account the fact that 22 bits of the key r are fixed at 0 effectively reducing the dimension of $\mathcal{S}$.

6 Proofs and Simulations

In this section we give some theoretical and simulation results that further support the claims we made thus far.

6.1 About the Fast Convolution Algorithm

Proof of query complexity for the claim made in Section 3.2. Consider, without loss of generality and for blocks of size n, that we possess $a \cdot 2^{2n/3}$ blocks of keystream and the same number of blocks of encrypted secret S with a a function of n. So in this setting we have $a^2 \cdot 2^{4n/3}$ different XORed-values possible between the two lists, that we will consider as independent and uniformly distributed over $2^n - 1$ values. We will then focus on the $2n/3$ bits truncation, $T(\cdot)$, and ignore the rest. We count the number of occurrences for every truncated values and store them in two lists of size $2^{2n/3}$. Using the fast Walsh-Hadamard transform 3 times, Algorithm 5, we can therefore compute the same counters but for all the XORed-values. We hope that the counter for $T(S)$, the good counter, will be lower than all of the other counters, the bad counters, with probability $\Omega(1)$. In which case we say the algorithm succeeds.

Let X_i^c represents the fact that the i^{th} value truncates to c, so that X_i^c follows a Bernoulli distribution and any counter can be written as $X^c = \sum_{i=1}^{a^2 2^{4n/3}} X_i^c$. Now we have to discriminate between the distributions of the good and bad counters:

$$\begin{aligned}
\text{Good case } c = T(S): \quad & \Pr(X_i^{T(S)} = 1) = (2^{n/3} - 1)/2^n = 2^{-2n/3} - 2^{-n} \\
& \implies \mathbf{E}[X^{T(S)}] = 2^{2n/3}a^2 - 2^{n/3}a^2 \\
\text{Bad case } c \neq T(S): \quad & \Pr(X_i^c = 1) = (2^{n/3})/2^n = 2^{-2n/3} \\
& \implies \mathbf{E}[X^c] = 2^{2n/3}a^2
\end{aligned}$$

Now we are interested by the probability that a bad counter gets a value below $\mathbf{E}[X^{T(S)}]$ as a measure of how distinct the distributions are. Using Chernov Bound we get for all $c \neq T(S)$:

$$\begin{aligned}
\Pr(X^c < \mathbf{E}[X^{T(S)}]) &= \Pr(X^c < (1 - 2^{-n/3})2^{2n/3}a^2) \\
&= \Pr(X^c < (1 - 2^{-n/3})\mathbf{E}[X^c]) \\
&\leq e^{-((2^{-n/3})^2 \cdot 2^{2n/3}a^2)/2)} = e^{-a^2/2}
\end{aligned}$$

And to compute the probability that no bad counter gets below $\mathbf{E}[X^{T(S)}]$ we will have to assume their independence, which is wrong, but we will come back later to discuss this assumption.

$$\begin{aligned}\Pr(\forall c \neq T(S) : X^c \geq \mathbf{E}[X^{T(S)}]) &= \prod_{c \neq T(S)} \Big(1 - \Pr(X^c < \mathbf{E}[X^{T(S)}])\Big) \\ &\geq \left(1 - e^{-a^2/2}\right)^{2^{2n/3}}\end{aligned}$$

To conclude, we need to find an $a = a(n)$ such that this probability remains greater than some positive value as n grows. This is clearly achieved with $a = \mathcal{O}(\sqrt{n})$ as for example taking $a = \frac{2\sqrt{n}}{\sqrt{3 \cdot \log_2(e)}} \simeq 0.96\sqrt{n}$ we get:

$$\begin{aligned}\Pr(\forall c \neq T(S) : X^c \geq \mathbf{E}[X^{T(S)}]) &\geq (1 - e^{-a^2/2})^{2^{2n/3}} \\ &\geq (1 - 2^{-2n/3})^{2^{2n/3}} \\ &\geq 0.25, \qquad\qquad \forall n \geq 3/2\end{aligned}$$

Therefore we can bound the probability of success by the events '$X^{T(S)} < \mathbf{E}[X^{T(S)}]$', probability $\simeq 1/2$, and '$\forall c \neq T(S) : X^c \geq \mathbf{E}[X^{T(S)}]$', probability at least $1/4$. Then we indeed have a probability of at least $1/8$ of having a successful algorithm. We can conclude that with $\mathcal{O}(n \cdot 2^{4n/3})$ XORed-values the algorithm has probability $\Omega(1)$ of succeeding.

Notice that this requires lists of size $\mathcal{O}(\sqrt{n} \cdot 2^{2n/3})$ but for the proof we only need the total number of pairs between the two lists. So we can break the requirement that the two lists are of comparable sizes as long as the product of their sizes sum up to the order of required values.

On the independence of the counters, this is obviously wrong as they are bound by the relation $\sum_c X^c = a^2 2^{4n/3}$. However this relation becomes looser and looser as n grows so the approximation obtained should still be correct asymptotically. Moreover, the covariances implied are negative *i.e.* knowing one draw is big makes the other draws smaller in expectation to compensate. Small negative covariances will make the distribution look more evenly distributed in the sense that we can't observe too many extreme events in a particular direction which is good for the success rate of the algorithm. So the assumption of independence may be a conservative one for this complexity analysis.

Simulation results. We ran simulations for block sizes $n = 12$, 24, 32 and 48 bits, so that we could do some statistical estimations of the success probability for this attack. We first create two lists of same size, one of raw keystream output and one XORed with an n-bit secret S. Then we pass the two lists in algorithm 5 counting over $n' = 2n/3$ bits (unless specified otherwise) to get a list of counters for each possible XOR outputs on those n' bits. Then the expected behaviour of the attack would be to look for a solution whose n' first bits correspond to the position of the lowest counter and test this hypothesis with algorithm 3. If it returns a unique value then this is S and we are done, if it returns an empty set then test with the position of the second lowest counter, etc. We can therefore

know the number of key candidates that would be required to recover S and, over many trials, have an estimation of the probability of success after a given number of candidates in these parameters.

For block sizes of 12 and 24 we simulated a permutation simply by shuffling a range into a list. For bigger sizes of 32 and 48 we used the Simon lightweight cipher from the NSA [3] as that is one of the rare block cipher who can act on 48-bit blocks. We could quickly gather 10 000 runs for each setting except for the 48-bit blocks simulation where we gathered 756 runs.

In general we observe in Figure 6 that the algorithm has a good chance of success with the first few candidates when using the suggested parameters. Moreover the sensibility with respect to the data complexity (Figure 4) and to the number of bits counted over (Figure 5) is fairly high. These results back up our complexity analysis and are a good indication that no big constant is ignored by the $\mathcal{O}()$ notation.

On the speed at which the probability increases we realized that, despite the log scale on the x axis, the curves take a straight (Figure 3) or concave shape (Figure 5 6). That means that the probability of success with the next key candidate decreases very quickly with the number of key candidates already tested and proved wrong. For example for $n = 48$ bits (Figure 6) over 756 trials the right key candidate was in the 2048 lowest counters in 98.1% of the time but the worst case found was 1 313 576 and these "very bad" cases push the mean rank of the right key candidate to 2287 and its sample variance to 2 336 937 008.

For $n = 48$ bits, one simulation took us 40 minutes over 10 cores (each step is highly parallelizable), and 64 gibibytes of RAM for the counters lists.

6.2 About the Known Prefix Sieving Algorithm

We consider two particular settings for the known prefix sieving algorithm and the corresping block splitting attack, with $z = n/2$ and $z = 1$.

Theoretical bound. We first give a theoretical lower bound to the probability of success of the sieving when $\dim(\mathcal{S}) = n/2$ (*i.e.* $z = n/2$), depending on the query complexity. Every partial collision found helps us to sieve. After collecting many blocks of keystream and encryption of S let $|\mathcal{A}| \cdot |\mathcal{B}| =: \alpha 2^n$ for some α. Thus we get $\alpha 2^n / 2^{n/2} = \alpha 2^{n/2}$ partial collisions in expectation. More precisely, the Chernoff bound gives us a lower bound for the probability of finding at least $(1-\delta)\alpha 2^{n/2}$ collisions:

$$p \geq 1 - \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}} \right)^{\alpha 2^{n/2}}$$

for any $\delta > 0$.

We see one partial collision as a draw in the coupon collector problem. One can use the formula in [34] for the tail of coupon collector problem probability

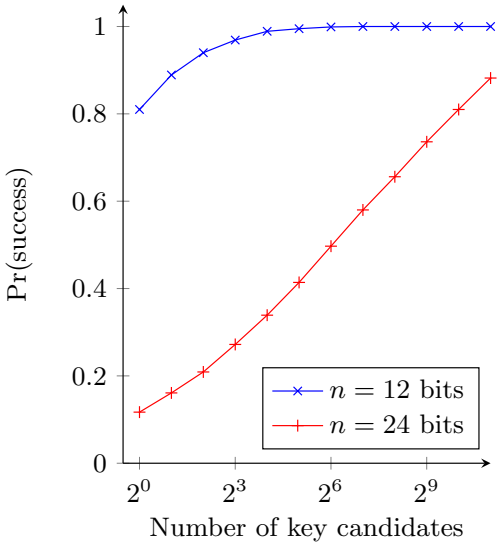


Fig. 3. Results for lists size of $3 \cdot 2^{2n/3}$

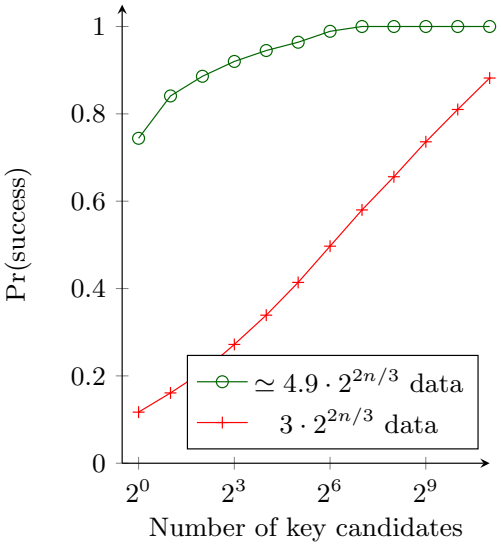


Fig. 4. Results for $n = 24$ bits

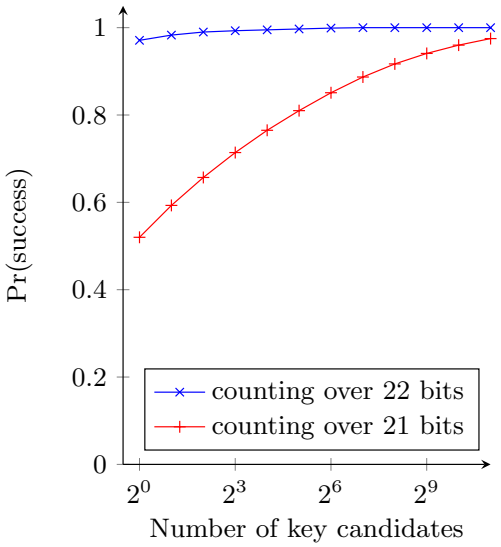


Fig. 5. Results for $n = 32$ bits; $\sqrt{n}2^{2n/3} \simeq 5.66 \cdot 2^{2n/3}$ data

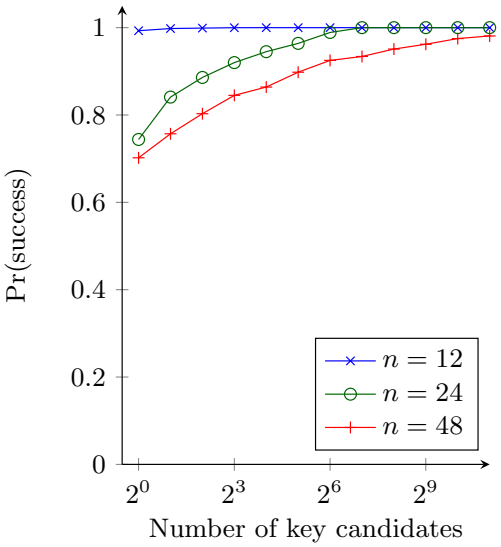


Fig. 6. Results for $\sqrt{n}2^{2n/3}$ data; counting over $2n/3$ bits

distribution to estimate the chance of success after obtaining $\beta \cdot 2^{n/2}$ partial collisions:

$$p \geq 1 - 2^{-\beta/\ln(2)+n/2}$$

which is positive whenever $\beta \geq n/2 \cdot \ln(2)$.

Therefore we bound the probability of success when collecting $|\mathcal{A}| \cdot |\mathcal{B}| = \alpha 2^n$ pairs as the probability of obtaining at least $(1-\delta)\alpha 2^{n/2}$ partial collisions multiplied by the probability of success after sieving $(1-\delta)\alpha 2^{n/2}$ values:

$$p \geq \left(1 - \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\alpha 2^{n/2}}\right) \cdot \left(1 - 2^{-(1-\delta)\cdot\alpha/\ln(2)+n/2}\right)$$

In particular, with two lists of size $\sqrt{n/2} \cdot 2^{n/2}$ (*i.e.* $\alpha = n/2$), we get $p \geq 0.99$ as long as $n \geq 32$ (using $\delta = 2^{-8}$).

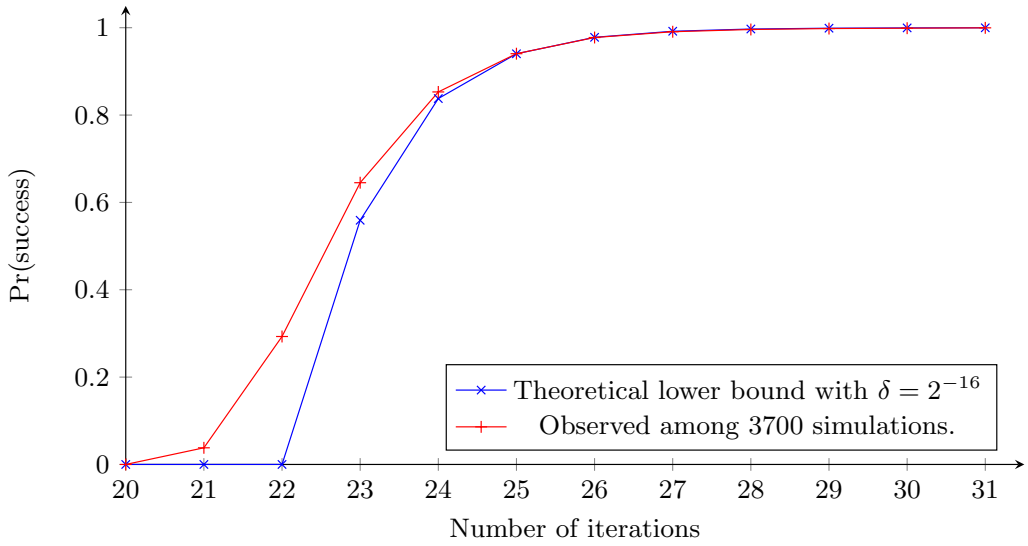


Fig. 7. Probability of success of the known prefix sieving knowing 2^{32} encryptions of a 32-bit secret against the number of chunks of 2^{32} keystream blocks of size $n = 64$ bits used.

Simulation results. We ran simulations with a block size $n = 64$ bits, and a secret S of size $n/2 = 32$ bits, using the Tiny Encryption Algorithm (TEA [37]) in CTR mode to encrypt the data. We create two lists, the keystream output list $a_i \in \mathcal{A}$, and the encryptions $b_j = a_j \oplus (\overline{0} \,\|\, S) \in \mathcal{B}$. We first produce and sort a list $\mathcal{B}$ with 2^{32} elements then produce, sort and sieve iteratively several lists $\mathcal{A}$ with 2^{32} elements, until the secret S is the only one remaining in the sieve.

One simulation runs in around 20 minutes over 36 cores, as every steps are trivially parallelizable: encryption, sorting and sieving. We ran 3700 simulations

and tracked how many chunks of $2^{n/2} = 2^{32}$ keystream outputs were needed for sieving. The coupon collector problem predicts that one will need on average $n/2 \cdot \ln(2) \cdot 2^{n/2}$ partial collisions which will be obtained after $n/2 \cdot \ln(2) \simeq 22.18 < 23$ rounds in expectation. And indeed the simulations showed a 64.5% probability of success after 23 iterations. Figure 7 shows the convergence between the theoretical lower bound and the simulated probabilities. We also noticed that the discrepancy in the number of rounds required is largely due to the last few candidates remaining in the sieve. If we decided the attack is successful when we are left with less than 1000 potential candidates for the secret then the algorithm successfully finishes after 16 rounds every time. In fact after 16 rounds the number of candidates left varies from 419 to 560 in all the simulations we have run.

Bit by bit secret recovery. We also want to study the complexity of recovering the secret S bit by bit as an extreme case of the block splitting scenario described in Section 4.2. For simplicity, we consider a setting where one query returns a block of keystream and the encryption of $0 \,\|\, s_i$ with an unknown bit s_i. We are interested in the query complexity for recovering n bits of secret one bit at a time; that is we need to know the first bit to ask for the second one, etc. Clearly this can be done in $\mathcal{O}(n \cdot 2^{n/2})$ queries by repeating n times the attack on one bit. But the intuition is that we may need less and less queries to uncover the next bit as we go forward and accumulate blocks of keystream.

Let:

$U_i \leftarrow$The expected number of encryption of $0 \,\|\, s_i$ to recover s_i.
$K_i \leftarrow$The expected number of raw keystream outputs to recover s_i.

From the definition of a query, the above description and because each time we find a bit of secret we can deduce a range of keystream blocks for the next step we have the relations:

$$K_1 = U_1 \tag{1}$$
$$K_{i+1} = K_i + U_i + U_{i+1} \quad \text{for } i \geq 1 \tag{2}$$
$$K_i \cdot U_i = 2^n \quad \text{(in expectation)} \tag{3}$$

We consider the following proposition:

$$P_i : U_i = 2^{n/2}(\sqrt{i} - \sqrt{i-1}),$$

and, using (2), when P_k true for all $k \leq i$ we have:

$$K_i = 2\sum_{k=1}^{i-1} U_k + U_i = 2^{n/2}(\sqrt{i} + \sqrt{i-1}).$$

Moreover (1) and (3) imply $K_1 = U_1 = 2^{n/2}$ so P_1 is true. Now suppose P_k true for all $k \leq i$, let's prove it holds for P_{i+1}:

$$\begin{aligned}
& K_{i+1} \cdot U_{i+1} = 2^n && \text{by (3)} \\
\Longrightarrow\ & U_{i+1}^2 + (K_i + U_i) \cdot U_{i+1} - 2^n = 0 && \text{by (2)} \\
\Longrightarrow\ & U_{i+1}^2 + 2^{n/2} \cdot 2\sqrt{i} \cdot U_{i+1} - 2^n = 0 && \text{by } P_i \\
\Longrightarrow\ & U_{i+1} = 2^{n/2}(\sqrt{i+1} - \sqrt{i}) && \text{as } U_{i+1} \geq 0 \\
\Longrightarrow\ & P_{i+1} \text{ is true.}
\end{aligned}$$

Now that we have a closed form for U_i we can deduce the expected number of queries needed to recover n bits of secret by summing over as $\sum_{i=1}^{n} U_i = 2^{n/2}\sqrt{n}$.

Therefore the query complexity is really $\mathcal{O}(\sqrt{n} \cdot 2^{n/2})$ ignoring a constant depending on the length of a query. Notice that this complexity is the same as when sieving S as a whole showing that we don't grow the query complexity by more than a constant with this strategy.

Conclusion

In this work, we have studied the missing difference problem and its relation to the security of the CTR mode. We have given efficient algorithms for the missing difference problem in two practically relevant cases: with an arbitrary missing difference, and when the missing difference is known to be in some low-dimension vector space. These algorithms lead to a message-recovery attack against the CTR mode with complexity $\tilde{\mathcal{O}}(2^{n/2})$, and a universal forgery attack against some Carter-Wegman MACs with complexity $\tilde{\mathcal{O}}(2^{2n/3})$.

In particular, we show that message-recovery attacks against the CTR mode can be mounted with roughly the same requirements and the same complexity as attacks against the CBC mode. While both modes have similar security proofs, there was a folklore assumption that the security loss of the CTR mode with large amounts of data is slower than in the CBC mode, because the absence of collision in the CTR keystream is harder to exploit than CBC collisions [15, Section 4.8.2]. Our results show that this is baseless, and use of the CTR mode with 64-bit block ciphers should be considered unsafe (unless strict data limits are in place). As a counter-measure, we recommend to use larger block sizes, and to rekey well before $2^{n/2}$ blocks of data. Concrete guidelines for 128-bit block ciphers have been given by Luykx and Paterson [26]. Alternatively, if the use of small block is required, we suggest using a mode with provable security beyond the birthday bound, such as CENC [21,22].

Our missing difference attacks against CTR and the collision attacks against CBC are two different possible failure of block cipher modes beyond the birthday bound. They exploit different properties of the modes but result in similar attacks. These techniques can be used against other modes of operations (OFB, CFB, ...), and most of them will be vulnerable to at least one the attacks, unless they have been specially designed to provide security beyond the birthday bound.

Acknowledgement

Part of this work was supported by the French DGA, and the authors are partially supported by the French Agence Nationale de la Recherche through the BRUTUS project under Contract ANR-14-CE28-0015.

References

1. Albrecht, M.R., Degabriele, J.P., Hansen, T.B., Paterson, K.G.: A surfeit of SSH cipher suites. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 16. pp. 1480–1491. ACM Press (Oct 2016)
2. AlFardan, N.J., Bernstein, D.J., Paterson, K.G., Poettering, B., Schuldt, J.C.N.: On the Security of RC4 in TLS. In: King, S.T. (ed.) USENIX Security 2013. pp. 305–320. USENIX Association (2013)
3. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: SIMON and SPECK: Block ciphers for the internet of things. Cryptology ePrint Archive, Report 2015/585 (2015), `http://eprint.iacr.org/2015/585`
4. Bellare, M., Kohno, T., Namprempre, C.: The Secure Shell (SSH) Transport Layer Encryption Modes. IETF RFC 4344 (2006)
5. Bellare, M., Desai, A., Jokipii, E., Rogaway, P.: A concrete security treatment of symmetric encryption. In: 38th FOCS. pp. 394–403. IEEE Computer Society Press (Oct 1997)
6. Bernstein, D.J.: The poly1305-AES message-authentication code. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 32–49. Springer, Heidelberg (Feb 2005)
7. Bhargavan, K., Leurent, G.: On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 16. pp. 456–467. ACM Press (Oct 2016)
8. Biham, E., Shamir, A.: Differential cryptanalysis of DES-like cryptosystems. Journal of Cryptology 4(1), 3–72 (1991)
9. Daemen, J., Knudsen, L.R., Rijmen, V.: The block cipher Square. In: Biham, E. (ed.) FSE'97. LNCS, vol. 1267, pp. 149–165. Springer, Heidelberg (Jan 1997)
10. Diffie, W., Hellman, M.E.: Privacy and authentication: An introduction to cryptography. Proceedings of the IEEE 67(3), 397–427 (1979)
11. Dinur, I., Leurent, G.: Improved generic attacks against hash-based MACs and HAIFA. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part I. LNCS, vol. 8616, pp. 149–168. Springer, Heidelberg (Aug 2014)
12. Duong, T., Rizzo, J.: Here come the $\oplus$ ninjas (2011)
13. Dworkin, M.: Recommendation for Block Cipher Modes of Operation: Methods and Techniques. NIST Special Publication 800-38A, National Institute for Standards and Technology (Dec 2001)
14. Ferguson, N.: Authentication weaknesses in GCM. Comment to NIST (2005), `http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/CWC-GCM/Ferguson2.pdf`
15. Ferguson, N., Schneier, B., Kohno, T.: Cryptography engineering: design principles and practical applications. John Wiley & Sons (2011)
16. DES modes of operation. NIST Special Publication 81, National Institute for Standards and Technology (Dec 1980)

17. Fuhr, T., Leurent, G., Suder, V.: Collision attacks against CAESAR candidates - forgery and key-recovery against AEZ and Marble. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015, Part II. LNCS, vol. 9453, pp. 510–532. Springer, Heidelberg (Nov / Dec 2015)
18. Guo, J., Peyrin, T., Sasaki, Y., Wang, L.: Updates on generic attacks against HMAC and NMAC. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part I. LNCS, vol. 8616, pp. 131–148. Springer, Heidelberg (Aug 2014)
19. Handschuh, H., Preneel, B.: Key-recovery attacks on universal hash function based MAC algorithms. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 144–161. Springer, Heidelberg (Aug 2008)
20. Hoang, V.T., Reyhanitabar, R., Rogaway, P., Vizár, D.: Online authenticated-encryption and its nonce-reuse misuse-resistance. In: Gennaro, R., Robshaw, M.J.B. (eds.) CRYPTO 2015, Part I. LNCS, vol. 9215, pp. 493–517. Springer, Heidelberg (Aug 2015)
21. Iwata, T.: New blockcipher modes of operation with beyond the birthday bound security. In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 310–327. Springer, Heidelberg (Mar 2006)
22. Iwata, T., Mennink, B., Vizár, D.: CENC is optimally secure. Cryptology ePrint Archive, Report 2016/1087 (2016), http://eprint.iacr.org/2016/1087
23. Joux, A.: Authentication failures in NIST version of GCM. Comment to NIST (2006), http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/800-38_Series-Drafts/GCM/Joux_comments.pdf
24. Lee, C., Kim, J., Sung, J., Hong, S., Lee, S.: Forgery and key recovery attacks on PMAC and mitchell's TMAC variant. In: Batten, L.M., Safavi-Naini, R. (eds.) ACISP 06. LNCS, vol. 4058, pp. 421–431. Springer, Heidelberg (Jul 2006)
25. Leurent, G., Peyrin, T., Wang, L.: New generic attacks against hash-based MACs. In: Sako, K., Sarkar, P. (eds.) ASIACRYPT 2013, Part II. LNCS, vol. 8270, pp. 1–20. Springer, Heidelberg (Dec 2013)
26. Luykx, A., Paterson, K.G.: Limits on authenticated encryption use in TLS (march 2016), http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf
27. Matsui, M.: Linear cryptoanalysis method for DES cipher. In: Helleseth, T. (ed.) EUROCRYPT'93. LNCS, vol. 765, pp. 386–397. Springer, Heidelberg (May 1994)
28. McGrew, D.: Impossible plaintext cryptanalysis and probable-plaintext collision attacks of 64-bit block cipher modes. Cryptology ePrint Archive, Report 2012/623. Accepted to FSE 2013. (2012), http://eprint.iacr.org/2012/623
29. McGrew, D.A., Viega, J.: The security and performance of the Galois/counter mode (GCM) of operation. In: Canteaut, A., Viswanathan, K. (eds.) INDOCRYPT 2004. LNCS, vol. 3348, pp. 343–355. Springer, Heidelberg (Dec 2004)
30. Peyrin, T., Wang, L.: Generic universal forgery attack on iterative hash-based MACs. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 147–164. Springer, Heidelberg (May 2014)
31. Preneel, B., van Oorschot, P.C.: MDx-MAC and building fast MACs from hash functions. In: Coppersmith, D. (ed.) CRYPTO'95. LNCS, vol. 963, pp. 1–14. Springer, Heidelberg (Aug 1995)
32. Preneel, B., van Oorschot, P.C.: On the security of two MAC algorithms. In: Maurer, U.M. (ed.) EUROCRYPT'96. LNCS, vol. 1070, pp. 19–32. Springer, Heidelberg (May 1996)
33. Procter, G., Cid, C.: On weak keys and forgery attacks against polynomial-based MAC schemes. Journal of Cryptology 28(4), 769–795 (Oct 2015)
34. Rajeev, M., Prabhakar, R.: Randomized Algorithms. Cambridge University Press (1995)

35. Rogaway, P.: Evaluation of some blockcipher modes of operation (2011)
36. Saarinen, M.J.O.: Cycling attacks on GCM, GHASH and other polynomial MACs and hashes. In: Canteaut, A. (ed.) FSE 2012. LNCS, vol. 7549, pp. 216–225. Springer, Heidelberg (Mar 2012)
37. Wheeler, D.J., Needham, R.M.: TEA, a tiny encryption algorithm. In: Preneel, B. (ed.) FSE'94. LNCS, vol. 1008, pp. 363–366. Springer, Heidelberg (Dec 1995)

Generic Attacks against Beyond-Birthday-Bound MACs

Gaëtan Leurent[1], Mridul Nandi[2], and Ferdinand Sibleyras[1]

[1] Inria, France
{gaetan.leurent,ferdinand.sibleyras}@inria.fr
[2] Indian Statistical Institute, Kolkata
mridul.nandi@gmail.com

Abstract. In this work, we study the security of several recent MAC constructions with provable security beyond the birthday bound. We consider block-cipher based constructions with a double-block internal state, such as SUM-ECBC, PMAC+, 3kf9, GCM-SIV2, and some variants (LightMAC+, 1kPMAC+). All these MACs have a security proof up to $2^{2n/3}$ queries, but there are no known attacks with less than 2^n queries.
We describe a new cryptanalysis technique for double-block MACs based on finding quadruples of messages with four pairwise collisions in halves of the state. We show how to detect such quadruples in SUM-ECBC, PMAC+, 3kf9, GCM-SIV2 and their variants with $\mathcal{O}(2^{3n/4})$ queries, and how to build a forgery attack with the same query complexity. The time complexity of these attacks is above 2^n, but it shows that the schemes do not reach full security in the information theoretic model. Surprisingly, our attack on LightMAC+ also invalidates a recent security proof by Naito.
Moreover, we give a variant of the attack against SUM-ECBC and GCM-SIV2 with time and data complexity $\tilde{\mathcal{O}}(2^{6n/7})$. As far as we know, this is the first attack with complexity below 2^n against a deterministic beyond-birthday-bound secure MAC.
As a side result, we also give a birthday attack against 1kf9, a single-key variant of 3kf9 that was withdrawn due to issues with the proof.

Keywords: Modes of operation, Cryptanalysis, Message Authentication Codes, Beyond-Birthday-Bound security

1 Introduction

Message authentication codes (or MACs) ensure the authenticity of messages in the secret-key setting. They are a core element of real-world security protocols such as TLS, SSH, or IPSEC. A MAC takes a message (and optionally a nonce) and a secret key to generate a tag that is sent with the message. Traditionally, they are classified into three types: deterministic, nonce-based, and probabilistic.

Deterministic MAC designs are the most popular, with widely used constructions based on block-cipher (CBC-MAC [13,4], OMAC [18], PMAC [5], LightMAC [29], ...) and hash functions (HMAC [2], NMAC [2], NI-MAC [1], ...). However, there is a generic forgery attack against all deterministic iterated MACs, using collisions in

the internal state, due to Preneel and van Oorschot [37]. Therefore, these MACs only achieve security up to the birthday bound, *i.e.* when the number of queries by the adversary is bounded by $2^{n/2}$, with n the state size. This is equivalently called $n/2$-bit security.

One way to increase the security is to use a *nonce*, a unique value provided by the user (in practice, the nonce is usually a counter). This approach has been pioneered by Wegman and Carter [41] based on an earlier work by Gilbert *et al.* [15]. Later a few follow ups like `EDM` and `EWCDM` [7], and `Dual EDM` [30] have been proposed to achieve beyond birthday security.

Alternatively, a probabilistic MAC uses a random coin for the extra value, which is usually called a *salt*, and must be transmitted with the MAC. Probabilistic MACs have the advantage that they can stay secure when called with the same input twice, and don't require a state to keep the nonce unique. Some popular probabilistic MAC constructions are `XMACR` [3], `RMAC` [22] and `EHtM` [31]. In particular, `RMAC` and `EHtM` have security beyond the birthday bound.

However, deterministic MACs are easier to use in practice, and there has been an important research effort to build deterministic MAC with security beyond the birthday bound, using an internal state larger than the primitive size. In particular, several constructions use a $2n$-bit internal state so that collisions in the state are only expected after 2^n queries. Yasuda first proposed `SUM-ECBC` [42], a beyond birthday bound (BBB) secure deterministic MAC that achieves $2n/3$-bit security. However, this construction has rate 1/2 and later Yasuda himself proposed one of the most popular BBB secure MAC `PMAC+` [43] achieving rate 1. Later several other constructions like `3kf9` [44], `LightMAC+` [33], `GCM-SIV2` [20], and single key `PMAC+` [9] have been proposed. Interestingly, all the above designs share a common structure: a double-block universal hash function outputs a $2n$-bit hash value (seen as two n-bit halves), and a finalization function generates the tag by XORing encrypted values of the two n-bit hash values. This structure has been called double-block-hash-then-sum, and it will be the focus of our paper.

More recently, variants of `PMAC+` based on tweakable block-cipher have also been proposed, such as `PMAC_TBC` [32], `PMACx` [27], `ZMAC` [21], and `ZMAC+`[28].

Our results. We focus on the security of deterministic block-cipher based MACs with security beyond the birthday bound and double-block hash construction. Several previous works have been focused on security proofs, showing that they are secure up to $2^{2n/3}$ queries [43,44,20,9,42,33]. For most of these constructions, the advantage of an adversary making q short queries is bounded by $\mathcal{O}(q^3/2^{2n})$. Recently, Naito [34] gave an improved security proof for `LighMAC+`, with advantage at most $\mathcal{O}(q_t^2 q_v/2^{2n})$, with q_t MAC queries and q_v verification queries. In particular, this would prove security up to 2^n when the adversary can only do a single verification query.

In this work, we take the opposite approach and look for generic attacks against these modes. We use a cryptanalysis technique that can be seen as a generalisation of the collision attack of Preneel and van Oorschot [37]. Instead of looking for a pair of messages so that the full state collides, we look for a quadruple of messages, which can be seen either as two pairs colliding on the first

Table 1. Summary of the security for studied modes and our main results. q is the number of queries, ℓ is maximum size of a query, σ is total number of processed blocks. The expected lower bound and attack complexity is in number of constant length queries ($\ell = \mathcal{O}(1)$). We use "U" for universal forgeries, and "E" for existential forgeries.

Mode	Provable security bounds		Attacks (this work)		
	Advantage	Queries	Queries	Time	Type
`SUM-ECBC` [42]	$\mathcal{O}(\frac{q^3\ell^3}{2^{2n}})$	$\Omega(2^{2n/3})$	$\mathcal{O}(2^{3n/4})$	$\tilde{\mathcal{O}}(2^{3n/2})$	U
			$\mathcal{O}(2^{6n/7})$	$\tilde{\mathcal{O}}(2^{6n/7})$	U
`GCM-SIV2` [20]	$\mathcal{O}(\frac{q^3\ell^2}{2^{2n}})$	$\Omega(2^{2n/3})$	$\mathcal{O}(2^{3n/4})$	$\tilde{\mathcal{O}}(2^{3n/2})$	U
			$\mathcal{O}(2^{6n/7})$	$\tilde{\mathcal{O}}(2^{6n/7})$	U
`PMAC+` [43]	$\mathcal{O}(\frac{q^3\ell^3}{2^{2n}})$	$\Omega(2^{2n/3})$	$\mathcal{O}(2^{3n/4})$	$\tilde{\mathcal{O}}(2^{3n/2})$	E
`LightMAC+` [33]	$\mathcal{O}(\frac{q^3}{2^{2n}})$	$\Omega(2^{2n/3})$	$\mathcal{O}(2^{3n/4})$	$\tilde{\mathcal{O}}(2^{3n/2})$	E
`1kPMAC+` [9]	$\mathcal{O}(\frac{\sigma}{2^n} + \frac{q\sigma^2}{2^{2n}})$	$\Omega(2^{2n/3})$	$\mathcal{O}(2^{3n/4})$	$\tilde{\mathcal{O}}(2^{3n/2})$	E
`3kf9` [44]	$\mathcal{O}(\frac{q^3\ell^3}{2^{2n}} + \frac{q\ell}{2^n})$	$\Omega(2^{2n/3})$	$\mathcal{O}(\sqrt[4]{n} \cdot 2^{3n/4})$	$\tilde{\mathcal{O}}(2^{5n/4})$	U
`1kf9` [8]	$\mathcal{O}(\frac{q\ell^2}{2^n} + \frac{q^3\ell^4}{2^{2n}} + \frac{q^4\ell^4}{2^{3n}} + \frac{q^4\ell^6}{2^{4n}})$	$\Omega(2^{2n/3})$	$\mathcal{O}(2^{n/2})$	$\tilde{\mathcal{O}}(2^{n/2})$	U

half of the state, or two pairs colliding on the second half. Since the finalization function combines the halves with a sum, we can detect such a quadruple because the corresponding MACs sum to zero, and can usually amplify this filtering. Moreover, when the message are well constructed, the relations defining the four collisions create a linear system of rank only three, so that we expect one good quadruple out of 2^{3n}. Therefore, we only need four lists of $2^{3n/4}$ queries, and we expect one good quadruple out of the 2^{3n} choices in the four lists.

Table 1 shows a summary of our main results and how they compare with their respective provable security claims. In particular, we have forgeries attacks with $\mathcal{O}(2^{3n/4})$ MAC queries against `SUM-ECBC`, `GCM-SIV2`, `PMAC+`, `LightMAC+`, `1kPMAC+`, and `3kf9`. As far as we know, these are the first attacks with less than 2^n queries against these constructions. Our attack against `LighMAC+` contradicts the recent security bound for `LighMAC+` [34], because we have an attack with $\mathcal{O}(2^{3n/4})$ MAC queries, and a single verification query. The other attacks do not contradict the security proofs, but they make an important step towards understanding the actually security of these modes: we now have a lower bound of $2^{2n/3}$ queries from the proofs, and an upper bound of $2^{3n/4}$ from our attacks.

The attacks have a complexity of $2^{3n/4}$ in the information theoretic model (the model used for most MAC security proofs), but we note that an attacker needs more than 2^n operations to create a forgery. However, we have found a variant of our attack against `SUM-ECBC` and `GCM-SIV2` with total complexity below 2^n, using $\mathcal{O}(2^{6n/7})$ queries and $\tilde{\mathcal{O}}(2^{6n/7})$ operations.

We have also found an attack with only $\mathcal{O}(2^{n/2})$ queries and $\tilde{\mathcal{O}}(2^{n/2})$ operations against `1kf9` [8], a single key variant of `3kf9` with claimed security up to $2^{2n/3}$ queries. `1kf9` has been withdrawn due to issues with its security proof, but no attack was known previously.

Related works. There has been extensive work on security proofs for modes of operations, with a recent focus on security beyond the birthday bound. An interesting example is the encryption mode `CENC` by Iwata [17]: the initial proof was only up to $2^{2n/3}$ queries, but a later proof showed that it actually remains secure close to 2^n queries [19]. Our results show that in the case of double-block-hash-then-sum MACs, the security is lower than n-bit security.

Similarly, the initial proof of the randomized MAC `EHtM` only gave security up to $2^{2n/3}$, but a later proof showed security up to $2^{3n/4}$ [11]. This result also includes a matching attack, using a technique similar to ours based on looking for quadruples. However in the case of `EHtM` the attacker can observe part of the state, which allows him to find a right quadruple in $\mathcal{O}(2^{3n/4})$ time and memory. In our case we can't observe the internal state at all, thus we need to use different tricks tailored to each construction in order to amplify the filtering and avoid the many false-positives. In particular, this significantly increases the time and memory complexity.

There has also been intensive work on generic attacks to complement the security proof results. After the generic collision attack of Preneel and van Oorschot [37], more advanced attacks against MACs have been described, with stronger outcomes than existential forgeries, starting with a key-recovery attack against the envelop MAC by the same authors [38]. In particular, a series of attacks against hash-based MACs [26,36,16,10] led to universal forgery attacks against long challenges, and key-recovery attacks when the hash function has an internal checksum (like the GOST family). Against `PMAC`, Lee *et al.* showed a universal forgery attack in 2006 [25]. Later, Fuhr, Leurent and Suder gave a key-recovery attack against the `PMAC` variant used in AEZv3 [14]. Issues with `GCM` authentication with truncated tags were also pointed out by Ferguson [12]. These attacks don't contradict the security proofs of the schemes, but they are important results to understand the security degradation after the birthday bound.

Organization of the paper. We first explain our attack technique using quadruples of messages in Section 2, and give three concrete attacks using this technique: an attack against `SUM-ECBC` and `GCM-SIV2` in Section 3, an attack against `PMAC+` and related constructions in Section 4, and an attack against `3kf9` in Section 5. Finally, we show a variant of the technique using special properties of the single-key constructions of [8,9] in Section 6.

Notations. We denote the concatenation of messages blocks x and y as $x \parallel y$. When x and y fit together in one block, we use $x|y$ to denote their concatenation. We use $L[i]$ to denote element i of list L, $x_{[i]}$ to denote bit i of x, and $x_{[i:j]}$ to denote bits i to $j-1$. Finally, we use a curly brace for systems of equations.

2 Generic Attack against double-block-hash MACs

We first explain our attacks in a generic way, and leave the specific details to later sections focused on concrete MAC constructions.

We consider MACs where the $2n$-bit internal state is divided in two n-bit parts, that we denote Σ and Θ, and the final MAC is computed as:

$$\mathrm{MAC}(M) = E\big(\Sigma(M)\big) \oplus E'\big(\Theta(M)\big),$$

where E and E' denote the block cipher with potentially different keys. The functions Σ and Θ can be seen as two n-bit universal hash functions computed on the message, hence the name double-block-hash-then-sum MAC.

Our attacks exploit the fact that the two halves are combined with a sum, where one side depends only on Σ, and the other side depends only on Θ. They do not seem applicable to constructions with more intricate finalization functions, such as `LightMAC+2` [33], or the tweakable block-cipher based constructions `PMAC_TBC` [32], `PMACx` [27], `ZMAC` [21], or `ZMAC+`[28].

2.1 Using Quadruples

Our strategy consists in looking for a quadruple of messages (X, Y, Z, T) such that pairs of values collide for one half of the state. More precisely, we look for quadruples satisfying a relation $\mathcal{R}(X, Y, Z, T)$ defined as:

$$\mathcal{R}(X, Y, Z, T) := \begin{cases} \Sigma(X) = \Sigma(Y) \\ \Theta(Y) = \Theta(Z) \\ \Sigma(Z) = \Sigma(T) \\ \Theta(T) = \Theta(X) \end{cases}$$

In particular, since the MAC is computed as $\mathrm{MAC}(M) = E\big(\Sigma(M)\big) \oplus E'\big(\Theta(M)\big)$, it follows that:

$$\mathcal{R}(X, Y, Z, T) \implies \mathrm{MAC}(X) \oplus \mathrm{MAC}(Y) \oplus \mathrm{MAC}(Z) \oplus \mathrm{MAC}(T) = 0. \qquad (1)$$

In addition, if the messages X, Y, Z, T are well constructed, the relation $\mathcal{R}$ reduces to a linear system of rank only three, *i.e.*

$$\big[\Sigma(X) = \Sigma(Y) \text{ and } \Theta(Y) = \Theta(Z) \text{ and } \Sigma(Z) = \Sigma(T)\big] \implies \Theta(T) = \Theta(X).$$

Therefore, we expect to find one quadruple satisfying the relation out of 2^{3n}, and we can construct 2^{3n} quadruples with just $4 \times 2^{3n/4}$ queries. This gives an attack with data complexity $\mathcal{O}(2^{3n/4})$.

In practice, we consider lists of $2^{3n/4}$ messages, generated with two message injection functions ϕ and ψ. These functions are different in every attack, but they mostly correspond to adding two distinct prefixes, as in the following example:

$$\begin{aligned} \phi(i) &= 0 \,\|\, i & \psi(i) &= 1 \,\|\, i \\ X = \phi(x) &= 0 \,\|\, x & Y = \psi(y) &= 1 \,\|\, y \\ Z = \phi(z) &= 0 \,\|\, z & T = \psi(t) &= 1 \,\|\, t, \end{aligned}$$

In particular, the pairs (X, Y), (Y, Z), (Z, T) and (T, X) that we consider always contain a message built with ϕ and message built with ψ. Therefore, we will have the required collisions in Σ or Θ if the difference introduced in the half-state by the second block cancels the difference found after processing the first block.

This type of attack has some similarities with a higher order differential attack. Indeed, in the easiest case (*e. g.* our attack against `SUM-ECBC`), the relation $\mathcal{R}$ can be written as $\mathcal{R}(x, y, z, t) \iff \left[x \oplus y = z \oplus t = \Delta_1 \text{ and } x \oplus t = y \oplus z = \Delta_3\right]$ for some secret values Δ_1 and Δ_3. This idea of looking for quadruples is also very similar to the attack on `EHtM` [11], but the full attack will turn out quite different. Indeed, in the case of `EHtM`, the attacker can observe the salt R which represent half of the $2n$-bit internal state. Here this would be the equivalent of observing $\Sigma(m)$ for all processed messages m. This is clearly not possible for the studied constructions and we need something more to discriminate and find a good quadruple that satisfies $\mathcal{R}$.

2.2 Detecting Quadruples: Generalized Birthday Algorithms

To finish the attack we usually need to locate one good quadruple. The relation $\mathrm{MAC}(X) \oplus \mathrm{MAC}(Y) \oplus \mathrm{MAC}(Z) \oplus \mathrm{MAC}(T) = 0$ in itself is too weak because we expect one quadruple out of 2^n to satisfy it randomly, but we can usually amplify the filtering using related quadruples that satisfy $\mathcal{R}$ simultaneously (the exact details depend on the MAC construction).

In most of our attacks, we can express the search for a quadruple as an instance of the 4-sum problem, and solve it using variants of Wagner's generalized birthday algorithm [40]. This reduces the time complexity of the attacks (compared to a naive search), and provides trade-offs between the query, memory and time complexities.

More precisely, our problem can be stated as follow:

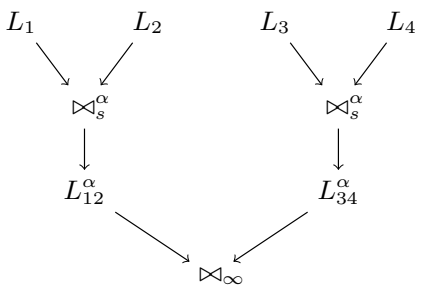
Definition 1 (4-sum problem). *Given four lists L_1, L_2, L_3, L_4 of 2^s elements, with on average 2^p quadruples $(x, y, z, t) \in L_1 \times L_2 \times L_3 \times L_4$ such that $x \oplus y \oplus z \oplus t = 0$, find one of them.*

Note that if the lists contain random n-bit words, we expect to have $2^p = 2^{4s-n}$ solutions, but in some of our instances there are more solutions because of the structure of the lists.

We denote the join operator as $\bowtie$; it computes the pairwise sum of two lists, and keeps the initial values attached to the sum. In addition, the join operator with filtering $\bowtie_t^\alpha$ only keeps values such that the t least significant bits of the sum agree with the value α:

$$A \bowtie B = \{(a \oplus b, a, b) \ : \ (a, b) \in A \times B\}$$
$$A \bowtie_t^\alpha B = \left\{(a \oplus b, a, b) \ : \ (a, b) \in A \times B,\ a_{[0:t]} \oplus b_{[0:t]} = \alpha\right\}$$

In particular, we have $\bowtie = \bowtie_0^0$. We also denote as $\bowtie_\infty$ the joint operator with filtering over the full input values. The filtered joint operator is the basis of Wagner's algorithm, and it can be computed in almost linear time by sorting the two input lists, and stepping through them simultaneously.



for $\alpha \in \{0,1\}^s$ **do**
  $L_{12}^\alpha \leftarrow L_1 \bowtie_s^\alpha L_2$
  $L_{34}^\alpha \leftarrow L_3 \bowtie_s^\alpha L_4$
  $L \leftarrow L_{12}^\alpha \bowtie_\infty L_{34}^\alpha$
  if $L \neq \varnothing$ **then**
    $(0,(x,y),(z,t)) \leftarrow L[0]$
    return (x,y,z,t)
  end if
end for
return $\perp$

Fig. 1. Generalized Birthday algorithm to find good quadruples.

Direct algorithm. While a naive algorithm for our 4-sum instances would take time 2^{4s} to examine all quadruples, there is a simple improvement with time and memory $\tilde{\mathcal{O}}(2^{2s})$. First, the attacker builds $L_{12} = L_1 \bowtie L_2$ and $L_{34} = L_3 \bowtie L_4$. Then, he looks for a collision between the first component of L_{12} and L_{34}. A collision directly yields a solution. This always finds a solution if it exists in $\tilde{\mathcal{O}}(2^{2s})$ operations but it also takes $\mathcal{O}(2^{2s})$ memory.

Memory efficient algorithm. We can reduce the memory complexity of the algorithm if we avoid constructing the full lists L_{12} and L_{34}. An algorithm with low memory complexity was first described by Chose *et al.* [6], but we use the description given by Wagner in the full version of [40].

Instead of building the full lists L_{12} and L_{34}, we filter values such that s least significant bits differ by some fixed value α. This reduces the expected size of the lists to only 2^s: $E[|L_{34}^\alpha|] = E[|L_{12}^\alpha|] = |L_1| \cdot |L_2|/2^s = 2^s$. If this algorithm is repeated for every s-bit value α, it will eventually find all solutions.

Actually, one run of the algorithm detects the solutions whose least significant bits of $x \oplus y$ are equal to α. If there are 2^p solutions in total, there is one such solution with probability 2^{p-s}, and this algorithm will find the first solution after trying 2^{s-p} values of α on average. Therefore, the expected time complexity of the algorithm given by Figure 1 is only $\tilde{\mathcal{O}}(2^{2s-p})$.

Related work. In a 2016 work, Nikolic and Sasaki [35] investigate the 4-sum where we need to find 4 different inputs x, y, z, t to a function f such that $f(x) \oplus f(y) \oplus f(z) \oplus f(t) = 0$. They also mention that their algorithm is adaptable to pairwise identical functions, *i. e.* $f(x) \oplus g(y) \oplus f(z) \oplus g(t) = 0$.

Most of our attacks can be written in this way; concretely, they are equivalent to instances of random functions with $3n$-bit outputs. In this setting our algorithm takes time $\tilde{\mathcal{O}}(2^{3n/2})$ and memory $\mathcal{O}(2^{3n/4})$, while Nikolic and Sasaki's work can reach $\tilde{\mathcal{O}}(2^{9n/8})$ time and $\mathcal{O}(2^{3n/4})$ memory. Unfortunately, their algorithm requires $\tilde{\mathcal{O}}(2^{9n/8})$ queries to the functions; this would translate to $\tilde{\mathcal{O}}(2^{9n/8})$ queries to the MAC, which is not interesting in our context.

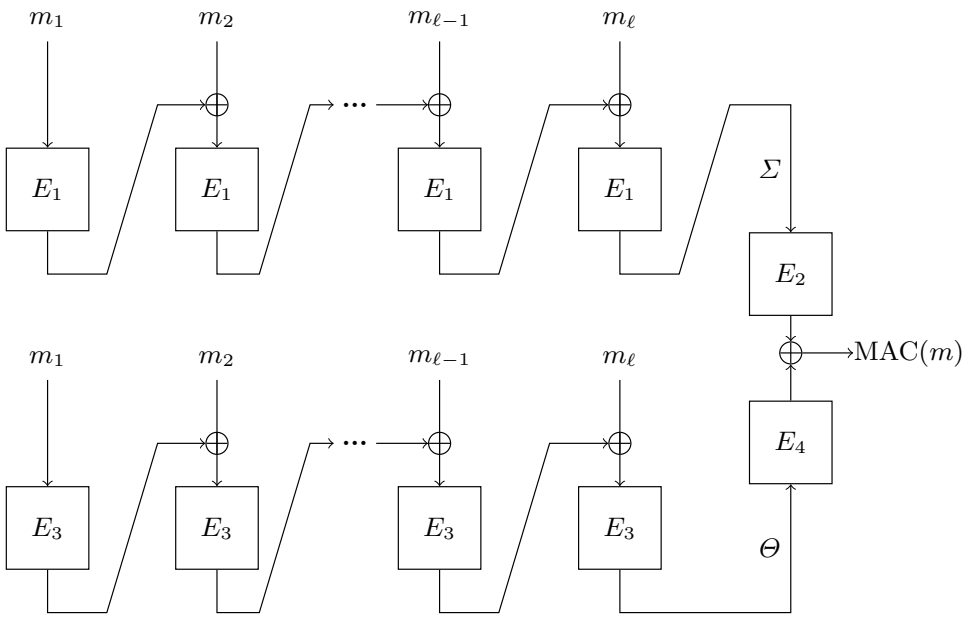


Fig. 2. Diagram for SUM-ECBC with a $\ell-$block message.

3 Attacking SUM-ECBC-like constructions

We start with attacks against SUM-ECBC [42] and GCM-SIV2 [20]; while the constructions are quite different, they have a similar structure and the same attacks can be used in both cases. We give a universal forgery attack with $\mathcal{O}(2^{3n/4})$ queries and $\tilde{\mathcal{O}}(2^{3n/2})$ operations (using memory $\mathcal{O}(2^{3n/4})$), and a variant with total complexity below 2^n, with $\mathcal{O}(2^{6n/7})$ queries and $\tilde{\mathcal{O}}(2^{6n/7})$ operations.

3.1 Attacking SUM-ECBC

SUM-ECBC was designed by Yasuda in 2010 [42], inspired by MAC constructions summing two CBC-MACs in the ISO 9797-1 standard. The scheme uses a block cipher keyed with four independent keys, denoted as E_1, E_2, E_3, E_4. The message M is first padded with 10^* padding, and divided into n-bit blocks. In the following we ignore the padding and consider the padded message as the input: this makes our description easier, and any padded message whose last block is non-zero can be "un-padded" to generate a valid input message. The construction is defined as follows (see also Figure 2):

$$\begin{aligned} \Sigma(M) &= \sigma_\ell & \sigma_0 &= 0 \quad & \sigma_i &= E_1(\sigma_{i-1} \oplus m_i) \\ \Theta(M) &= \theta_\ell & \theta_0 &= 0 \quad & \theta_i &= E_3(\theta_{i-1} \oplus m_i) \\ \mathrm{MAC}(M) &= E_2(\Sigma(M)) \oplus E_4(\Theta(M)) \end{aligned}$$

Attack. Following the framework of Section 2, we consider quadruple of messages, built with two message injection functions:

$$\phi(i) = 0 \,\|\, i \qquad\qquad \psi(i) = 1 \,\|\, i$$

In particular, we have

$$\mathrm{MAC}(\phi(i)) = E_2\Big(\underbrace{E_1\big(i \oplus E_1(0)\big)}_{\Sigma_0(i)}\Big) \oplus E_4\Big(\underbrace{E_3\big(i \oplus E_3(0)\big)}_{\Theta_0(i)}\Big)$$
$$\mathrm{MAC}(\psi(i)) = E_2\Big(\underbrace{E_1\big(i \oplus E_1(1)\big)}_{\Sigma_1(i)}\Big) \oplus E_4\Big(\underbrace{E_3\big(i \oplus E_3(1)\big)}_{\Theta_1(i)}\Big)$$

Next, we build quadruples of messages X, Y, Z, T with

$$X = \phi(x) \qquad Y = \psi(y) \qquad Z = \phi(z) \qquad T = \psi(t),$$

and we look for a quadruple with partial state collisions for the underlying pairs, *i. e.* a quadruple following the relation:

$$\mathcal{R}(x, y, z, t) := \begin{cases} \Sigma_0(x) = \Sigma_1(y) \\ \Sigma_0(z) = \Sigma_1(t) \\ \Theta_0(z) = \Theta_1(y) \\ \Theta_0(x) = \Theta_1(t). \end{cases}$$

We have

$$\mathcal{R}(x, y, z, t) \Leftrightarrow \begin{cases} x \oplus E_1(0) = y \oplus E_1(1) \\ z \oplus E_3(0) = y \oplus E_3(1) \\ z \oplus E_1(0) = t \oplus E_1(1) \\ x \oplus E_3(0) = t \oplus E_3(1) \end{cases} \Leftrightarrow \begin{cases} x \oplus y \oplus z \oplus t = 0 \\ x \oplus y = E_1(0) \oplus E_1(1) \\ x \oplus t = E_3(0) \oplus E_3(1) \end{cases}$$

As promised in Section 2, $\mathcal{R}$ defines a $3n-$bit relation. We can easily observe when $x \oplus y \oplus z \oplus t = 0$, and we can also detect the relation on the sum of the MACs following Equation (1):

$$\mathcal{R}(x, y, z, t) \Rightarrow \mathrm{MAC}(\phi(x)) \oplus \mathrm{MAC}(\psi(y)) \oplus \mathrm{MAC}(\phi(z)) \oplus \mathrm{MAC}(\psi(t)) = 0$$

Moreover, we observe that $\mathcal{R}(x, y, z, t)$ is satisfied if and only if $\mathcal{R}(x \oplus c, y \oplus c, z \oplus c, t \oplus c)$ is satisfied for any constant c. We use this relation to build several quadruples that satisfy $\mathcal{R}$ simultaneously:

$$\mathcal{R}(x, y, z, t) \iff \forall c,\, \mathcal{R}(x \oplus c, y \oplus c, z \oplus c, t \oplus c) \tag{2}$$

This leads to an attack with $\mathcal{O}(2^{3n/4})$ queries: we consider four sets $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{T}$ of $2^{3n/4}$ values, and we look for a quadruple $(x, y, z, t) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{Z} \times \mathcal{T}$ with:

$$\begin{cases} x \oplus y \oplus z \oplus t = 0 \\ \mathrm{MAC}(\phi(x)) \oplus \mathrm{MAC}(\psi(y)) \oplus \mathrm{MAC}(\phi(z)) \oplus \mathrm{MAC}(\psi(t)) = 0 \\ \mathrm{MAC}(\phi(x \oplus 1)) \oplus \mathrm{MAC}(\psi(y \oplus 1)) \oplus \mathrm{MAC}(\phi(z \oplus 1)) \oplus \mathrm{MAC}(\psi(t \oplus 1)) = 0. \end{cases} \tag{3}$$

Because we need a fair distribution of values $x \oplus y$ and $x \oplus t$ to find the good quadruple we build the sets as:

$$\mathcal{X} = \left\{x \in \{0,1\}^n \ : \ x_{[0:n/4]} = 0\right\} \qquad \mathcal{Y} = \left\{x \in \{0,1\}^n \ : \ x_{[n/4:n/2]} = 0\right\}$$
$$\mathcal{Z} = \left\{x \in \{0,1\}^n \ : \ x_{[n/2:3n/4]} = 0\right\} \qquad \mathcal{T} = \left\{x \in \{0,1\}^n \ : \ x_{[3n/4:n]} = 0\right\}$$

With this construction, there is exactly one quadruple $(x, y, z, t) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{Z} \times \mathcal{T}$ that respects $\mathcal{R}$, given by:

$$x = v_1|w_2|u_3|0 \qquad y = w_1|v_2|0|u_4 \qquad z = u_1|0|v_3|w_4 \qquad t = 0|u_2|w_3|v_4,$$

where:

$$\begin{aligned} E_1(0) \oplus E_1(1) &=: u_1|u_2|u_3|u_4 \\ E_3(0) \oplus E_3(1) &=: v_1|v_2|v_3|v_4 \\ E_1(0) \oplus E_1(1) \oplus E_3(0) \oplus E_3(1) &=: w_1|w_2|w_3|w_4. \end{aligned}$$

We expect on average one random quadruple satisfying (3) (with 2^{3n} potential quadruples, and a $3n$-bit filtering), in addition to the quadruple satisfying $\mathcal{R}$. The correct quadruple can easily be checked with a few extra queries.

In practice, we use the generalized birthday algorithms of Section 2.2 in order to optimize the complexity of the attack. We consider four lists:

$$\begin{aligned} L_1 &= \{x \parallel \mathrm{MAC}(\phi(x)) \parallel \mathrm{MAC}(\phi(x \oplus 1)) \ : \ x \in \mathcal{X}\} \\ L_2 &= \{y \parallel \mathrm{MAC}(\psi(y)) \parallel \mathrm{MAC}(\psi(y \oplus 1)) \ : \ y \in \mathcal{Y}\} \\ L_3 &= \{z \parallel \mathrm{MAC}(\phi(z)) \parallel \mathrm{MAC}(\phi(z \oplus 1)) \ : \ z \in \mathcal{Z}\} \\ L_4 &= \{t \parallel \mathrm{MAC}(\psi(t)) \parallel \mathrm{MAC}(\psi(t \oplus 1)) \ : \ t \in \mathcal{T}\} \end{aligned}$$

Notice that we can build those lists with $5 \cdot 2^{3n/4}$ queries as, by construction, for any element i of $\mathcal{Y}, \mathcal{Z}, \mathcal{T}$ the element $(i \oplus 1)$ also belongs to $\mathcal{Y}, \mathcal{Z}, \mathcal{T}$, respectively. We use the algorithm of Section 2.2 to find $(x, y, z, t) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{Z} \times \mathcal{T}$ such that $L_1[x] \oplus L_2[y] \oplus L_3[z] \oplus L_4[t] = 0$ with $\tilde{\mathcal{O}}(2^{3n/2})$ operations, using a memory of size $\mathcal{O}(2^{3n/4})$. After finding a collision, we verify that it is not a false positive by testing the relation for another value c. As there are on average $\mathcal{O}(1)$ random quadruples the attack is indeed using a total of $5 \cdot 2^{3n/4} + \mathcal{O}(1) = \mathcal{O}(2^{3n/4})$ queries.

Universal Forgeries. This attack can be extended to a universal forgery. Indeed, the fixed prefix 0 and 1 can be replaced by v and $v \oplus 1$ for any block v, and when we identify a right quadruple (x, y, z, t) we deduce the value $\Delta_1 = E_1(v) \oplus E_1(v \oplus 1)$ and $\Delta_3 = E_3(v) \oplus E_3(v \oplus 1)$. There is also a length extension property: if (x, y, z, t) is a right quadruple, then $\mathrm{MAC}(v \parallel x \parallel s) \oplus \mathrm{MAC}(v \oplus 1 \parallel y \parallel s) \oplus \mathrm{MAC}(v \parallel z \parallel s) \oplus \mathrm{MAC}(v \oplus 1 \parallel t \parallel s) = 0$ for any suffix s.

Therefore if we want to forge a MAC for any message m of size $\ell \geq 2$ blocks we parse it as $m = v \parallel w \parallel s$ (where s has zero, one, or several blocks) and perform

the attack to recover Δ_1 and Δ_3. Then we can forge using the previous relation, and Equation (2):

$$\begin{aligned}\mathrm{MAC}(v \parallel w \parallel s) = &\mathrm{MAC}(v \oplus 1 \parallel w \oplus \Delta_1 \parallel s) \oplus \mathrm{MAC}(v \parallel w \oplus \Delta_3 \parallel s) \\ &\oplus \mathrm{MAC}(v \oplus 1 \parallel w \oplus \Delta_1 \oplus \Delta_3 \parallel s)\end{aligned}$$

Optimizing the time complexity. Equation (2) can also be used to reduce the time complexity below 2^n, at the cost of more oracle queries. Indeed, if we consider a subset $\mathcal{C}$ of $\{0,1\}^n$, we have:

$$\begin{aligned}\mathcal{R}(x,y,z,t) &\Leftrightarrow \forall c \in \mathcal{C}, \mathcal{R}(x \oplus c, y \oplus c, z \oplus c, t \oplus c) \\ &\Rightarrow \forall c \in \mathcal{C},\ \mathrm{MAC}(\phi(x \oplus c)) \oplus \mathrm{MAC}(\psi(y \oplus c)) \\ &\qquad \oplus \mathrm{MAC}(\phi(z \oplus c)) \oplus \mathrm{MAC}(\psi(t \oplus c)) = 0 \\ &\Rightarrow \textstyle\bigoplus_{c\in\mathcal{C}} \mathrm{MAC}(\phi(x \oplus c)) \oplus \bigoplus_{c\in\mathcal{C}} \mathrm{MAC}(\psi(y \oplus c)) \\ &\qquad \oplus \textstyle\bigoplus_{c\in\mathcal{C}} \mathrm{MAC}(\phi(z \oplus c)) \oplus \bigoplus_{c\in\mathcal{C}} \mathrm{MAC}(\psi(t \oplus c)) = 0 \quad (4)\end{aligned}$$

If we select $\mathcal{C}$ as a linear subspace, then the last expression does not depend on the full (x,y,z,t), but only on their projection on the orthogonal of $\mathcal{C}$. Concretely, we use $\mathcal{C} = \left\{x : x_{[3n/7:n]} = 0\right\} = \left\{x : x < 2^{3n/7}\right\}$, so that the value $\bigoplus_{c\in\mathcal{C}} \mathrm{MAC}(\phi(x \oplus c))$ is independent of bits 0 to $3n/7 - 1$ of x.

Therefore, we consider the rewritten MAC function

$$\mathrm{MAC}'(v \parallel w) = \textstyle\bigoplus_{c\in\mathcal{C}} \mathrm{MAC}(v \parallel w \oplus c),$$

the following message injections, with a $4n/7$-bit input

$$\phi'(i) = 0 \parallel i|0 \qquad\qquad \psi'(i) = 1 \parallel i|0,$$

and a reduced relation over $4n/7$-bit values:

$$\begin{aligned}\mathcal{R}'(x,y,z,t) &:= \begin{cases} x \oplus y = (E_1(0) \oplus E_1(1))_{[3n/7:n]} \\ y \oplus z = (E_3(0) \oplus E_3(1))_{[3n/7:n]} \\ z \oplus t = (E_1(0) \oplus E_1(1))_{[3n/7:n]} \\ t \oplus x = (E_3(0) \oplus E_3(1))_{[3n/7:n]} \end{cases} \\ &\Leftrightarrow \begin{cases} x \oplus y \oplus z \oplus t = 0 \\ x \oplus y = (E_1(0) \oplus E_1(1))_{[3n/7:n]} \\ x \oplus t = (E_3(0) \oplus E_3(1))_{[3n/7:n]} \end{cases}\end{aligned}$$

Thanks to Equation 4, we still have:

$$\mathcal{R}'(x,y,z,t) \Rightarrow \mathrm{MAC}'(\phi'(x)) \oplus \mathrm{MAC}'(\psi'(y)) \oplus \mathrm{MAC}'(\phi'(z)) \oplus \mathrm{MAC}'(\psi'(t)) = 0$$

Since the relation $\mathcal{R}'$ is now only a $12n/7$-bit condition, we can use shorter lists than before, with just $2^{3n/7}$ elements. We can also increase the filtering using

the same trick as previously, considering the following lists:

$$L'_1 = \left\{ x \,\|\, \mathrm{MAC}'(\phi'(x)) \,\|\, \mathrm{MAC}'(\phi'(x \oplus 1)) \;:\; x \in \{0,1\}^{4n/7},\, x_{[0:n/7]} = 0 \right\}$$
$$L'_2 = \left\{ y \,\|\, \mathrm{MAC}'(\psi'(y)) \,\|\, \mathrm{MAC}'(\psi'(y \oplus 1)) \;:\; y \in \{0,1\}^{4n/7},\, y_{[n/7:2n/7]} = 0 \right\}$$
$$L'_3 = \left\{ z \,\|\, \mathrm{MAC}'(\phi'(z)) \,\|\, \mathrm{MAC}'(\phi'(z \oplus 1)) \;:\; z \in \{0,1\}^{4n/7},\, z_{[2n/7:3n/7]} = 0 \right\}$$
$$L'_4 = \left\{ t \,\|\, \mathrm{MAC}'(\psi'(t)) \,\|\, \mathrm{MAC}'(\psi'(t \oplus 1)) \;:\; t \in \{0,1\}^{4n/7},\, t_{[3n/7:4n/7]} = 0 \right\}$$

Finally, using the algorithm of Section 2.2 with $s = 3n/7$ and $p = 0$, we can locate a right quadruple using $\tilde{\mathcal{O}}(2^{6n/7})$ queries, $\tilde{\mathcal{O}}(2^{6n/7})$ operations, and $\mathcal{O}(2^{3n/7})$ memory. This recovers only $4n/7$ bits of $E_1(0) \oplus E_1(1)$ and $E_3(0) \oplus E_3(1)$, but we can easily recover the remaining bits, either by brute force, or by repeating the attack with a different set $\mathcal{C}$.

3.2 Attacking GCM-SIV2

GCM-SIV2 is an authenticated encryption mode designed by Iwata and Minematsu [20] as a double-block-hash version of GCM-SIV (in the following, we consider GCM-SIV2 with GHASH as the underlying universal hash function). For simplicity, we focus on the authentication part of GCM-SIV2, using inputs with a non-empty associated data, and an empty message. In this case, GCM-SIV2 becomes a nonce-based MAC. The message M (considered as associated data for the mode) is zero-padded, divided into n-bit blocks, and the length is appended in an extra block. Then the construction is defined as follows, with $\odot$ a finite field multiplication (see also Figure 3):

$$\begin{aligned} \Sigma(N, M) &= N \oplus \ell \odot H_1 \oplus \textstyle\bigoplus_{i=1}^{\ell} m_i \odot H_1^{\ell+2-i} \\ \Theta(N, M) &= N \oplus \ell \odot H_2 \oplus \textstyle\bigoplus_{i=1}^{\ell} m_i \odot H_2^{\ell+2-i} \\ \mathrm{MAC}(N, M) &= E_1(\Sigma(M)) \oplus E_2(\Theta(M)) \,\big\|\, E_3(\Sigma(M)) \oplus E_4(\Theta(M)) \end{aligned}$$

Attack. The structure of the authentication part of GCM-SIV2 is essentially the same as the structure of SUM-ECBC, where the block cipher calls E_1 and E_3 are replaced by multiplication by H_1 and H_2. The finalization function has a $2n$-bit output $\mathrm{MAC}^1, \mathrm{MAC}^2$, but quadruples following $\mathcal{R}$ will collide on both outputs. Thus, we can essentially repeat the SUM-ECBC attack, but there is an important difference: GCM-SIV2 is a nonce-based MAC, rather than a deterministic one. Therefore, all queries must include a nonce N, and we should not query two different messages with the same nonce. We adapt the previous attack using message injection functions that output both a nonce and a message, so that we use two fixed messages, 0 and 1, with variable nonces:

$$\phi(i) = (i, 0) \qquad\qquad \psi(i) = (i, 1)$$

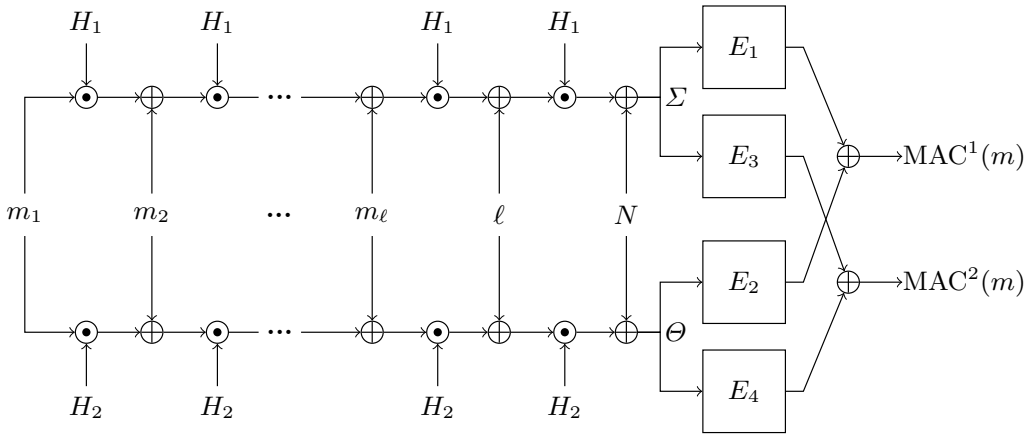


Fig. 3. Diagram for authentication in `GCM-SIV2` using `GHASH` with a ℓ-block message, a nonce N, hash keys H_1 and H_2.

$$\mathrm{MAC}(\phi(i)) = E_1\big(\underbrace{i \oplus H_1}_{\Sigma_0(i)}\big) \oplus E_2\big(\underbrace{i \oplus H_2}_{\Theta_0(i)}\big) \;\Big\|\; E_3\big(\Sigma_0(i)\big) \oplus E_4\big(\Theta_0(i)\big)$$

$$\mathrm{MAC}(\psi(i)) = E_1\big(\underbrace{i \oplus H_1 \oplus H_1^2}_{\Sigma_1(i)}\big) \oplus E_2\big(\underbrace{i \oplus H_2 \oplus H_2^2}_{\Theta_1(i)}\big) \;\Big\|\; E_3\big(\Sigma_1(i)\big) \oplus E_4\big(\Theta_1(i)\big).$$

We consider quadruples of nonce/messages X, Y, Z, T with

$$X = \phi(x) \qquad Y = \psi(y) \qquad Z = \phi(z) \qquad T = \psi(t),$$

and we have the same kind of relations as in the previous attack:

$$\mathcal{R}(x,y,z,t) := \begin{cases} \Sigma_0(x) = \Sigma_1(y) \\ \Sigma_0(z) = \Sigma_1(t) \\ \Theta_0(z) = \Theta_1(y) \\ \Theta_0(x) = \Theta_1(t). \end{cases} \Leftrightarrow \begin{cases} x \oplus y \oplus z \oplus t = 0 \\ x \oplus y = H_1^2 \\ x \oplus t = H_2^2 \end{cases}$$
$$\Rightarrow \mathrm{MAC}(\phi(x)) \oplus \mathrm{MAC}(\psi(y)) \oplus \mathrm{MAC}(\phi(z)) \oplus \mathrm{MAC}(\psi(t)) = 0$$

Since the MAC output is $2n$-bit long, we can directly build an attack with $\mathcal{O}(2^{3n/4})$ queries: we consider four distinct sets $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{T}$ of $2^{3n/4}$ values, and we look for a quadruple $(x, y, z, t) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{Z} \times \mathcal{T}$, such that

$$\begin{cases} x \oplus y \oplus z \oplus t = 0 \\ \mathrm{MAC}(\phi(x)) \oplus \mathrm{MAC}(\psi(y)) \oplus \mathrm{MAC}(\phi(z)) \oplus \mathrm{MAC}(\psi(t)) = 0 \end{cases} \tag{5}$$

we expect to find one good quadruple that respects $\mathcal{R}$ along with $\mathcal{O}(1)$ quadruples that randomly satisfy the observable filter (5). This leads to an attack with $\mathcal{O}(2^{3n/4})$ queries and time $\tilde{O}(2^{3n/2})$. Since we recover H_1 and H_2 (from $H_1^2 = x \oplus y$ and $H_2^2 = x \oplus t$), we can do universal forgeries. In addition, we can also easily adapt the attack with $\mathcal{O}(2^{6n/7})$ queries and time $\tilde{O}(2^{6n/7})$.

4 Attacking PMAC-like constructions

We now describe attacks against PMAC+ [43] and related constructions: 1kMAC+ [9], and LightMAC+ [33]. We have an existential forgery attack with $\mathcal{O}(2^{3n/4})$ queries and $\tilde{\mathcal{O}}(2^{3n/2})$ operations (using memory $\mathcal{O}(2^{3n/4})$), with a range of time-memory trade-offs with $\mathcal{O}(2^t)$ queries, with $3n/4 < t < n$, and $\tilde{\mathcal{O}}(2^{3n-2t})$ operations (using memory $\mathcal{O}(2^t)$).

4.1 Attacking PMAC+

PMAC+ was designed by Yasuda in 2011 [43], as a variant of PMAC [5] with a larger internal state. The scheme internally uses a tweakable block cipher construction inspired by the XE construction [39], that we denote as $\tilde{E}_i$. The message M is first padded with 10^* padding, and divided into n-bit blocks, but for simplicity we ignore the padding in our description. The construction is shown in Figure 4[3]:

$$\begin{aligned} \Sigma(M) &= \textstyle\bigoplus_{i=1}^{\ell} \tilde{E}_i(m_i) & \tilde{E}_i(x) &= E_1(x \oplus 2^i \odot \Delta_0 \oplus 2^{2i} \odot \Delta_1) \\ \Theta(M) &= \textstyle\bigoplus_{i=1}^{\ell} 2^{\ell-i} \odot \tilde{E}_i(m_i) & \Delta_0 &= E_1(0) \quad \Delta_1 = E_1(1) \\ \mathrm{MAC}(M) &= E_2(\Sigma(M)) \oplus E_3(\Theta(M)) \end{aligned}$$

Attack. As in the previous attack, we use message injection functions with two different prefixes, but we include an extra block u to define related quadruples:

$$\phi_u(i) = u \,\|\, 0 \,\|\, i \qquad\qquad \psi_u(i) = u \,\|\, 1 \,\|\, i$$

$$\mathrm{MAC}(\phi_u(i)) = E_2\Big(\underbrace{\tilde{E}_1(u) \oplus \tilde{E}_2(0) \oplus \tilde{E}_3(i)}_{\Sigma_{u,0}(i)}\Big) \oplus E_3\Big(\underbrace{4\tilde{E}_1(u) \oplus 2\tilde{E}_2(0) \oplus \tilde{E}_3(i)}_{\Theta_{u,0}(i)}\Big)$$

$$\mathrm{MAC}(\psi_u(i)) = E_2\Big(\underbrace{\tilde{E}_1(u) \oplus \tilde{E}_2(1) \oplus \tilde{E}_3(i)}_{\Sigma_{u,1}(i)}\Big) \oplus E_3\Big(\underbrace{4\tilde{E}_1(u) \oplus 2\tilde{E}_2(1) \oplus \tilde{E}_3(i)}_{\Theta_{u,1}(i)}\Big).$$

Next, we build quadruples of messages X, Y, Z, T with

$$X = \phi_u(x) \qquad Y = \psi_u(y) \qquad Z = \phi_u(z) \qquad T = \psi_u(t),$$

and we look for a quadruple with partial state collisions for the underlying pairs, *i. e.* a quadruple following the relation:

$$\mathcal{R}(x, y, z, t) := \begin{cases} \Sigma_{u,0}(x) = \Sigma_{u,1}(y) \\ \Sigma_{u,0}(z) = \Sigma_{u,1}(t) \\ \Theta_{u,0}(z) = \Theta_{u,1}(y) \\ \Theta_{u,0}(x) = \Theta_{u,1}(t). \end{cases}$$

[3] The algorithm and the figure given in [43] differ in the coefficients used to compute Θ. We use the algorithmic description because it matches later PMAC+ variants, but the attack can easily be adapted to the other case.

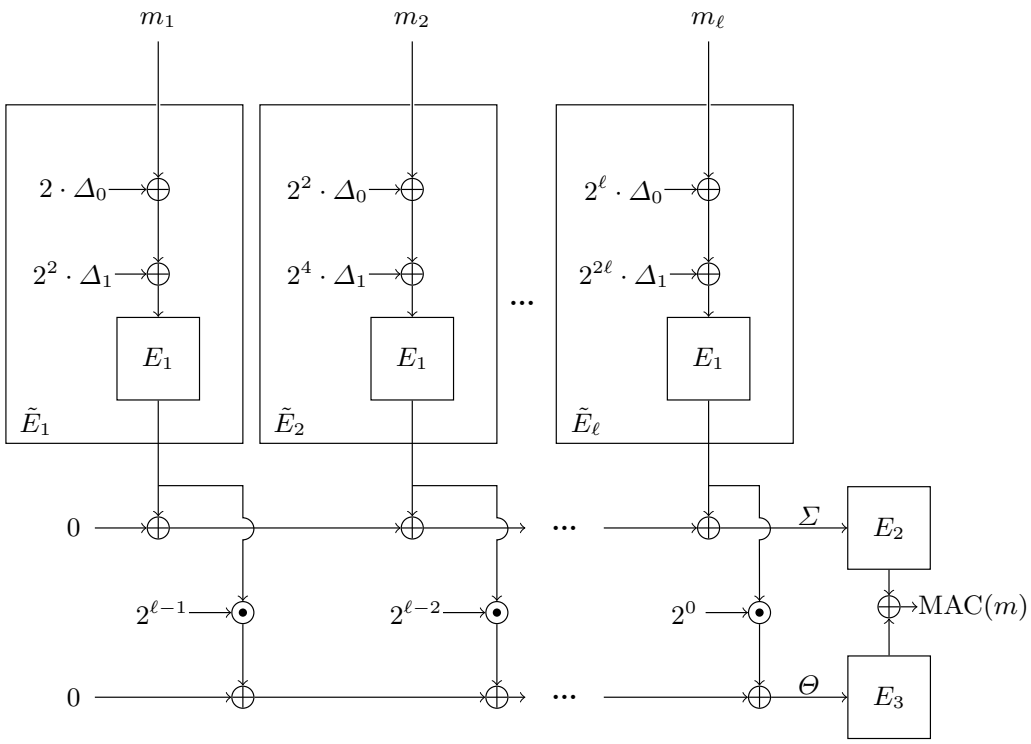


Fig. 4. Diagram for PMAC+ with a ℓ-block message where $\Delta_0 = E_1(0)$ and $\Delta_1 = E_1(1)$.

We have

$$\mathcal{R}(x,y,z,t) \Leftrightarrow \begin{cases} \tilde{E}_3(x) \oplus \tilde{E}_2(0) = \tilde{E}_3(y) \oplus \tilde{E}_2(1) \\ \tilde{E}_3(z) \oplus \tilde{E}_2(0) = \tilde{E}_3(t) \oplus \tilde{E}_2(1) \\ \tilde{E}_3(y) \oplus 2\tilde{E}_2(1) = \tilde{E}_3(z) \oplus 2\tilde{E}_2(0) \\ \tilde{E}_3(t) \oplus 2\tilde{E}_2(1) = \tilde{E}_3(x) \oplus 2\tilde{E}_2(0) \end{cases}$$

$$\Leftrightarrow \begin{cases} \tilde{E}_3(x) \oplus \tilde{E}_3(y) \oplus \tilde{E}_3(z) \oplus \tilde{E}_3(t) = 0 \\ \tilde{E}_3(x) \oplus \tilde{E}_3(y) = \tilde{E}_2(0) \oplus \tilde{E}_2(1) \\ \tilde{E}_3(t) \oplus \tilde{E}_3(x) = 2\tilde{E}_2(0) \oplus 2\tilde{E}_2(1) \end{cases}$$

Again, $\mathcal{R}$ defines a $3n-$bit relation, and we can detect it through the sum of the MACs following Equation (1):

$$\mathcal{R}(x,y,z,t) \Rightarrow \mathrm{MAC}(\phi_u(x)) \oplus \mathrm{MAC}(\psi_u(y)) \oplus \mathrm{MAC}(\phi_u(z)) \oplus \mathrm{MAC}(\psi_u(t)) = 0$$

In addition, the relation $\mathcal{R}$ is independent of the value u, so that we can easily build several quadruples that satisfy $\mathcal{R}$ simultaneously. This leads to an attack with $\mathcal{O}(2^{3n/4})$ queries: we consider four sets $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{T}$ of $2^{3n/4}$ random values, and we look for a quadruple $(x,y,z,t) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{Z} \times \mathcal{T}$, such that

$$\forall u \in \{0,1,2\},\ \mathrm{MAC}(\phi_u(x)) \oplus \mathrm{MAC}(\psi_u(y)) \oplus \mathrm{MAC}(\phi_u(z)) \oplus \mathrm{MAC}(\psi_u(t)) = 0$$

We expect on average one random quadruple (with 2^{3n} potential quadruples, and a $3n$-bit filtering), and one quadruple satisfying $\mathcal{R}$ (also a $3n$-bit condition). The correct quadruple can easily be checked with a few extra queries.

In practice, we use the generalized birthday algorithms of Section 2.2 in order to optimize the complexity of the attack. We consider four lists:

$$\begin{aligned}
L_1 &= \{\mathrm{MAC}(\phi_0(x)) \,\|\, \mathrm{MAC}(\phi_1(x)) \,\|\, \mathrm{MAC}(\phi_2(x)) \;:\; x \in \mathcal{X}\} \\
L_2 &= \{\mathrm{MAC}(\psi_0(y)) \,\|\, \mathrm{MAC}(\psi_1(y)) \,\|\, \mathrm{MAC}(\psi_2(y)) \;:\; y \in \mathcal{Y}\} \\
L_3 &= \{\mathrm{MAC}(\phi_0(z)) \,\|\, \mathrm{MAC}(\phi_1(z)) \,\|\, \mathrm{MAC}(\phi_2(z)) \;:\; z \in \mathcal{Z}\} \\
L_4 &= \{\mathrm{MAC}(\psi_0(t)) \,\|\, \mathrm{MAC}(\psi_1(t)) \,\|\, \mathrm{MAC}(\psi_2(t)) \;:\; t \in \mathcal{T}\}
\end{aligned}$$

and we look for a quadruple $(x, y, z, t) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{Z} \times \mathcal{T}$ such that $L_1[x] \oplus L_2[y] \oplus L_3[z] \oplus L_4[t] = 0$. This can be done with $\tilde{\mathcal{O}}(2^{3n/2})$ operations, using a memory of size $\mathcal{O}(2^{3n/4})$. Finally, once a quadruple (x, y, z, t) satisfying $\mathcal{R}(x, y, z, t)$ has been detected, it can be used to generate forgeries. Indeed, we can predict the MAC of a new message by making three new queries using Equation (1):

$$\forall u,\ \mathrm{MAC}(\phi_u(x)) = \mathrm{MAC}(\psi_u(y)) \oplus \mathrm{MAC}(\psi_u(z)) \oplus \mathrm{MAC}(\phi_u(t))$$

Time-Query Trade-offs. As opposed to the `SUM-ECBC` attack, we don't have an analogue to Equation (2) that can be used to reduce the time complexity. However, the time complexity of the algorithm can be slightly reduced when using more than $\mathcal{O}(2^{3n/4})$ queries. If we consider sets $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{T}$ of size 2^t with $3n/4 < t < n$, the resulting 4-sum is slightly easier, because there are 2^{4t-3n} expected solutions. Using the algorithm of section 2.2, this can be solved in time $\tilde{\mathcal{O}}(2^{3n-2t})$, using a memory of size $\mathcal{O}(2^t)$.

4.2 Attacking `LightMAC+`

`LightMAC+` was designed by Naito [33] using ideas from `PMAC+` [43] and `LightMAC` [29]. If we consider it as based on a tweakable block cipher $\tilde{E}$, it follows the same structure as `PMAC+` (see Figure 5), but $\tilde{E}$ takes a message block smaller than n bits:

$$\begin{aligned}
\Sigma(M) &= \textstyle\bigoplus_{i=1}^{\ell} \tilde{E}_i(m_i) & \tilde{E}_i(x) &= E_1(i|x) \\
\Theta(M) &= \textstyle\bigoplus_{i=1}^{\ell} 2^{\ell-i} \odot \tilde{E}_i(m_i) \\
\mathrm{MAC}(M) &= E_2(\Sigma(M)) \oplus E_3(\Theta(M))
\end{aligned}$$

Since the structure of `LightMAC+` is the same as the structure of `PMAC+`, we can use the same attack. The only difference from our point of view is that the message blocks are shorter than the block-size. As long as one message block is big enough to fit $2^{3n/4}$ different values, our attack will succeed.

This attack violates the improved security proof recently published at CT-RSA [34], with a security bound of $\mathcal{O}(q_t^2 q_v / 2^{2n})$ (with q_t MAC queries and q_v verification queries). Indeed, our attack reaches a constant success probability with $q_t = \mathcal{O}(2^{3n/4})$ and $q_v = 1$. We have shared our attack with Naito and he agreed that his proof is flawed.

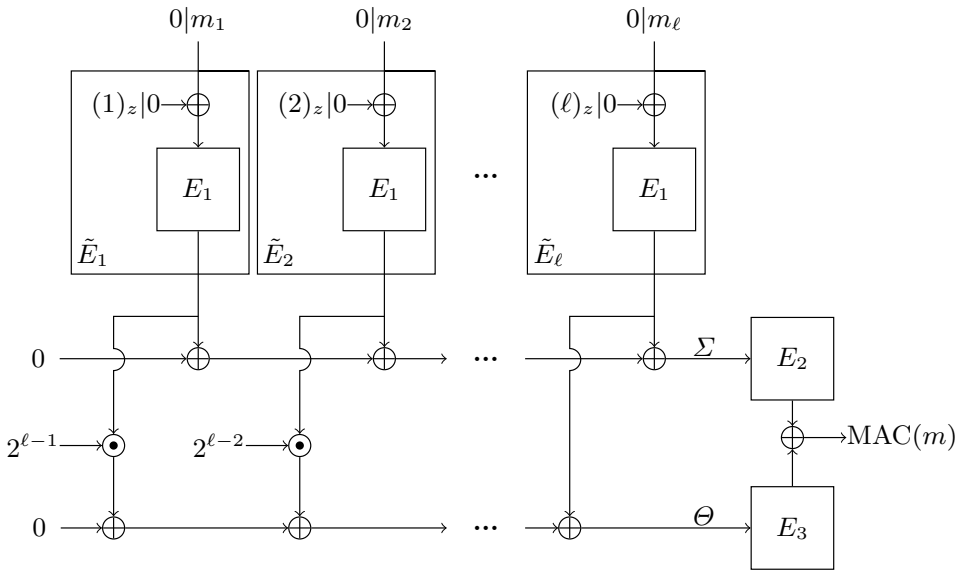


Fig. 5. Diagram for `LightMAC+` with $(n-z)$-bit blocks of a ℓ-block message where $(v)_z$ is the value v written over z bits.

4.3 Attacking `1kPMAC+`

`1kPMAC+` is a single-key variant of `PMAC+` [43] designed by Datta, Dutta, Nandi, Paul and Zhang [9], shown in Figure 8.

Since the structure of `1kPMAC+` is the same as the structure of `PMAC+`, we can use the same attack. Alternatively, we can take advantage of the `fix` functions to mount a more straightforward attack, as shown in Section 6.

5 Attacking `f9`-like constructions

Our third attack is applicable to `3kf9` [44] and similar constructions. We have a universal forgery attack with $\mathcal{O}(2^{3n/4})$ queries and $\tilde{\mathcal{O}}(2^{5n/4})$ operations using memory $\mathcal{O}(2^n)$, with a possible time-memory trade-offs.

5.1 Attacking `3kf9`

`3kf9` [44], designed by Xhang, Wu, Sui and Wang, is a three-key variant of the `f9` mode used in 3G telephony. While the original `f9` does not have security beyond the birthday bound [24], `3kf9` is secure up to $2^{2n/3}$ queries. We describe `3kf9` in Figure 6:

$$\Sigma(M) = \sigma_\ell \qquad \sigma_0 = 0 \qquad \sigma_i = E_1(\sigma_{i-1} \oplus m_i)$$
$$\Theta(M) = \bigoplus_{i=1}^{\ell} \sigma_i$$
$$\mathrm{MAC}(M) = E_2(\Sigma(M)) \oplus E_3(\Theta(M))$$

Attack. Our attack follows the same structure as the previous attacks. We start with messages of the form:

$$\phi(i) = 0 \,\|\, i \qquad\qquad \psi(i) = 1 \,\|\, i,$$

and the corresponding MACs:

$$\mathrm{MAC}(\phi(i)) = E_2\Big(\underbrace{E_1\big(x \oplus E_1(0)\big)}_{\Sigma_0(x)}\Big) \oplus E_3\Big(\underbrace{E_1\big(x \oplus E_1(0)\big) \oplus E_1\big(0\big)}_{\Theta_0(x)}\Big)$$
$$\mathrm{MAC}(\psi(i)) = E_2\Big(\underbrace{E_1\big(x \oplus E_1(1)\big)}_{\Sigma_1(x)}\Big) \oplus E_3\Big(\underbrace{E_1\big(x \oplus E_1(1)\big) \oplus E_1\big(1\big)}_{\Theta_1(x)}\Big).$$

We use quadruples of messages X, Y, Z, T with

$$X = \phi(x) \qquad Y = \psi(y) \qquad Z = \phi(z) \qquad T = \psi(t),$$

and we look for a quadruple with partial state collisions for the underlying pairs, *i. e.* a quadruple following the relation:

$$\begin{aligned}
\mathcal{R}(x,y,z,t) := &\begin{cases} \Sigma_0(x) = \Sigma_1(y) \\ \Sigma_0(z) = \Sigma_1(t) \\ \Theta_0(z) = \Theta_1(y) \\ \Theta_0(x) = \Theta_1(t). \end{cases} \\
\Leftrightarrow &\begin{cases} x \oplus E_1(0) = y \oplus E_1(1) \\ z \oplus E_1(0) = t \oplus E_1(1) \\ E_1(z \oplus E_1(0)) \oplus E_1(0) = E_1(y \oplus E_1(1)) \oplus E_1(1) \\ E_1(x \oplus E_1(0)) \oplus E_1(0) = E_1(t \oplus E_1(1)) \oplus E_1(1) \end{cases} \\
\Leftrightarrow &\begin{cases} x \oplus y \oplus z \oplus t = 0 \\ x \oplus y = E_1(0) \oplus E_1(1) \\ E_1(x \oplus E_1(0)) \oplus E_1(t \oplus E_1(1)) = E_1(0) \oplus E_1(1) \end{cases} \\
\Rightarrow &\ \mathrm{MAC}(\phi(x)) \oplus \mathrm{MAC}(\psi(y)) \oplus \mathrm{MAC}(\phi(z)) \oplus \mathrm{MAC}(\psi(t)) = 0.
\end{aligned}$$

As in the previous attacks, $\mathcal{R}$ defines a $3n-$bit relation. Moreover, we can easily observe when $x \oplus y \oplus z \oplus t = 0$, and the relation $x \oplus y = E_1(0) \oplus E_1(1)$ can be verified across several quadruples. We don't have related quadruples satisfying $\mathcal{R}$ simultaneously as in the previous attacks, but we can use those properties to detect right quadruples. This leads to an attack with $\tilde{\mathcal{O}}(2^{3n/4})$ queries: we consider four sets $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{T}$ of $\sqrt[4]{n} \times 2^{3n/4}$ random values, and we look for quadruples $(x, y, z, t) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{Z} \times \mathcal{T}$, such that:

$$\begin{cases} x \oplus y \oplus z \oplus t = 0 \\ \mathrm{MAC}(\phi(x)) \oplus \mathrm{MAC}(\psi(y)) \oplus \mathrm{MAC}(\phi(z)) \oplus \mathrm{MAC}(\psi(t)) = 0. \end{cases} \tag{6}$$

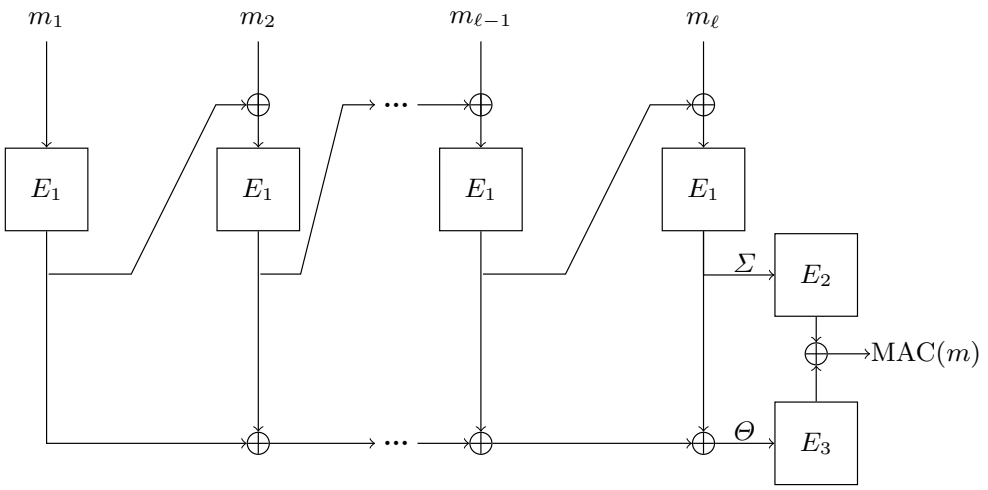


Fig. 6. Diagram for `3kf9` with a $\ell-$block message.

Since this a $2n$-bit condition, we expect on average $n \cdot 2^n$ quadruples (x, y, z, t) satisfying (6). In order to filter out the right ones, we look at the value $x \oplus y$ for all these quadruples. While the wrong quadruples should have a random $x \oplus y$, the right ones have $x \oplus y = E_1(0) \oplus E_1(1)$. Therefore, with high probability, the most frequent value for $x \oplus y$ is equal to $E_1(0) \oplus E_1(1)$, and quadruples satisfying this extra relation are right quadruples with probability $1/2$. More precisely, we expect on average n wrong quadruples for each value of $x \oplus y$, and n right quadruples with $x \oplus y = E_1(0) \oplus E_1(1)$.

Optimizing the time complexity. While the algorithm of Section 2.2 would take time $\tilde{\mathcal{O}}(2^{3n/2})$ with $\tilde{\mathcal{O}}(2^{3n/4})$ queries, we can reduce the time complexity using sets $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{T}$ with some structure. More precisely, we use:

$$\mathcal{X} = \mathcal{Z} = \left\{x \in \{0,1\}^n \ : \ x_{[0:n/4]} = 0\right\}$$
$$\mathcal{Y} = \mathcal{T} = \left\{x \in \{0,1\}^n \ : \ x_{[n/4:n/2]} = 0\right\}$$

so that quadruples can be written as

$$x =: x_3|x_2|x_1|0 \in \mathcal{X} \qquad y =: y_3|y_2|0|y_0 \in \mathcal{Y}$$
$$z =: z_3|z_2|z_1|0 \in \mathcal{Z} \qquad t =: t_3|t_2|0|t_0 \in \mathcal{T}.$$

In particular, right quadruples satisfy $x \oplus y \oplus z \oplus t = 0$, therefore $x_1 = z_1$, $y_0 = t_0$, and $x_3|x_2 \oplus z_3|z_2 = y_3|y_2 \oplus t_3|t_2$. We use these properties to adapt the algorithm of Section 2.2 and locate the quadruples efficiently. First we guess the $n/2$-bit value $\alpha_3|\alpha_2 := x_3|x_2 \oplus z_3|z_2 = y_3|y_2 \oplus t_3|t_3$. Then, for each $x = x_3|x_2|x_1|0$, there is a single candidate $z = (x_3 \oplus \alpha_3)|(x_2 \oplus \alpha_2)|x_1|0$ that could be part of a right quadruple. Similarly, every $y = y_3|y_2|0|y_0$ can be paired with a single

$t = (y_3 \oplus \alpha_3)|(y_2 \oplus \alpha_2)|0|y_0$. Therefore, we consider the two following lists:

$$L_1 = \{\mathrm{MAC}(\phi(x_3|x_2|x_1|0)) \oplus \mathrm{MAC}((x_3 \oplus \alpha_3)|(x_2 \oplus \alpha_2)|x_1|0) \ : \ x_3|x_2|x_1|0 \in \mathcal{X}\}$$
$$L_2 = \{\mathrm{MAC}(\phi(y_3|y_2|0|y_0)) \oplus \mathrm{MAC}((y_3 \oplus \alpha_3)|(y_2 \oplus \alpha_2)|0|y_0) \ : \ y_3|y_2|0|y_0 \in \mathcal{Y}\}$$

After sorting the lists, we look for matches, and the corresponding quadruples x, y, z, t are exactly the quadruples satisfying

$$\begin{cases} x \oplus y \oplus z \oplus t = 0 \\ (x \oplus z)_{[n/2:n]} = \alpha_3|\alpha_2 \\ \mathrm{MAC}(\phi(x)) \oplus \mathrm{MAC}(\psi(y)) \oplus \mathrm{MAC}(\phi(z)) \oplus \mathrm{MAC}(\psi(t)) = 0. \end{cases} \tag{7}$$

More precisely, a match $L_1[x] = L_2[y]$ suggests $z = x \oplus \alpha_3|\alpha_2|0|0$ and $t = y \oplus \alpha_3|\alpha_2|0|0$, but there are four corresponding quadruples: (x, y, z, t), (z, y, x, t), (x, t, z, y), (z, t, x, y), and two candidate values for $E_1(0) \oplus E_1(1)$: $x \oplus y$ and $x \oplus y \oplus \alpha_3|\alpha_2|0|0$.

We need $\tilde{\mathcal{O}}(2^{3n/4})$ operations to generate those quadruples. We repeat this $2^{n/2}$ times to exhaust all $n/2$-bit values $\alpha_3|\alpha_2$ and generate all quadruples satisfying (6). Finally, we use an array to count the number of occurrences of each possible value of $x \oplus y$. Each counter receives an average two values, but the counter corresponding to $E_1(0) \oplus E_1(1)$ will receive three values on average. After repeating all the operations $\mathcal{O}(n)$ times, with some arbitrary constants in place of the zero bits, the highest counter corresponds to $E_1(0) \oplus E_1(1)$ with high probability, as proved in Section 5.2. This gives an attack with $\tilde{\mathcal{O}}(2^{3n/4})$ queries, $\tilde{\mathcal{O}}(2^{5n/4})$ operations, and $\mathcal{O}(2^n)$ memory[4].

Time-Memory Trade-offs. We can reduce the memory usage if we store only a subset of the counters, and repeat the whole algorithm until the whole set has been covered. Concretely, we store only the counters with a fixed value for bits $[0 : n/8]$ and $[n/4 : 3n/8]$ of $x \oplus y$. Because of the way the lists L_1 and L_2 are constructed, we have actually fixed $n/8$ bits of y_0 and x_1, and we can reduce the lists to size $2^{5n/8}$. Therefore we evaluate $2^{3n/4}$ counters in time $\tilde{\mathcal{O}}(2^{n/2} \cdot 2^{5n/8})$, using only $\mathcal{O}(2^{3n/4})$ memory. We repeat iteratively over the full counter set, so we need time $\tilde{\mathcal{O}}(2^{n/4} \cdot 2^{n/2} \cdot 2^{5n/8}) = \tilde{\mathcal{O}}(2^{11n/8})$. More generally, we have a time-memory trade-off with time $\tilde{\mathcal{O}}(2^{5n/4+t/2})$ and memory $\mathcal{O}(2^{n-t})$ for $0 < t < n/4$.

Forgeries. Once we found a quadruple (x, y, z, t) that respects $\mathcal{R}(x, y, z, t)$ we know that after processing message $\phi(x) = 0 \parallel x$ and $\psi(t) = 1 \parallel t$, there is no difference in the Θ part of the state ($\Theta_0(x) = \Theta_1(t)$). Moreover we have $\Theta_0(x) = \Sigma_0(x) \oplus E_1(0)$ and $\Theta_1(t) = \Sigma_1(x) \oplus E_1(1)$; this implies that there is a difference $E_1(0) \oplus E_1(1) = x \oplus y$ in the Σ part of the state. Therefore, we can build a full state collision with message $0 \parallel x \parallel 0$ and $1 \parallel t \parallel x \oplus y$. In particular, the following relation can be used to create forgeries with an arbitrary message m (of any length):

$$\mathrm{MAC}(0 \parallel x \parallel 0 \parallel m) = \mathrm{MAC}(1 \parallel t \parallel x \oplus y \parallel m).$$

[4] We can actually reduce the polynomial factors by fixing only $(n - \log_2(n))/4$ bits to zero, in order to have sets of size $\sqrt[4]{n} \cdot 2^{3n/4}$.

Universal Forgeries. We can even forge the tag of an arbitrary message of length at least $(2n+2)$ blocks with complexity only $n+1$ times the complexity of the simple forgery attack. The technique is more advanced and inspired by the multi-collision attack described by Joux [23]. For ease of notation we'll show how to forge the signature for a message starting with $2n+2$ blocks of zero, but this can be trivially adapted for any message.

First, we find a quadruple (x_1, y_1, z_1, t_1) as before. Then we consider messages $0\|0$ and $1\|x_1 \oplus y_1$. Since $x_1 \oplus y_1 = E_1(0) \oplus E_1(1)$, we have $\Sigma(0\|0) = \Sigma(1\|x_1 \oplus y_1)$, *i. e.* the Σ part of the state collides. Moreover, we know the difference in the Θ part: $\Theta(0 \parallel 0) \oplus \Theta(1 \parallel x_1 \oplus y_1) = x_1 \oplus y_1$.

More generally, at step i we use message injection functions

$$\phi_i(x) = \underbrace{0 \parallel 0 \parallel \ldots \parallel 0}_{\times 2(i-1)} \parallel 0 \parallel x \qquad \psi_i(x) = \underbrace{0 \parallel 0 \parallel \ldots \parallel 0}_{\times 2(i-1)} \parallel 1 \parallel x,$$

to look for a quadruple of messages

$$X_i = \phi_i(x_i) \qquad Y_i = \psi_i(y_i) \qquad Z_i = \phi_i(z_i) \qquad T_i = \psi_i(t_i).$$

When a right quadruple (x_i, y_i, z_i, t_i) has been identified, we can deduce that the MACs for $0 \parallel 0 \parallel \ldots \parallel 0 \parallel 0 \parallel 0$ and $0 \parallel 0 \parallel \ldots \parallel 0 \parallel 1 \parallel x_i \oplus y_i$ will match on the Σ branch and differ by $x_i \oplus y_i$ in their Θ branch.

After several iterations, we have actually built a multi-collision: all the messages $h_1 \parallel h_2 \parallel \ldots \parallel h_n \parallel h_{n+1}$ with $h_i \in \{(1 \parallel x_i \oplus y_i), (0 \parallel 0)\}$ collide on the Σ branch. In addition, we also know the difference in the Θ branch for those messages: it is equal to $\bigoplus_{\{i\,:\,h_i \neq 0\|0\}} (x_i \oplus y_i)$.

After at most $n+1$ steps, we can find a non empty subset $\mathcal{I} \subseteq [1 : n+1]$ such that $\bigoplus_{i \in \mathcal{I}} (x_i \oplus y_i) = 0$ by simple linear algebra[5]. This gives a collision on the full state, using messages $m_0 = 0 \parallel 0 \parallel \ldots \parallel 0$ (with $2(n+1)$ blocks) and $h = h_1 \parallel h_2 \parallel \ldots \parallel h_n \parallel h_{n+1}$ with $h_i = 1 \parallel x_i \oplus y_i$ if $i \in \mathcal{I}$, $h_i = 0 \parallel 0$ otherwise. Since the full state collides, we have for any message m (of any length):

$$\mathrm{MAC}(h \parallel m) = \mathrm{MAC}(m_0 \parallel m).$$

5.2 Detailed Complexity Analysis

We want to prove the claim that one will need to find $\mathcal{O}(n \cdot 2^n)$ quadruples in order to finish the attack on `3kf9` described in Section 5.1. We say the attack finishes when we recover the target value $T = E(0) \oplus E(1)$.

Assuming that each quadruple we find respects $\mathcal{R}$ with probability $1/2^n$, we fill a list of counters for every suspected values of T; a random quadruple gives two random values and a right one gives one value equal to T and one random value. Therefore we sum up the distribution of an observable value x as:

$$x \begin{cases} \xleftarrow{\$} \{0,1\}^n & \text{with probability } 1 - 1/2^{n+1} \\ \longleftarrow T & \text{with probability } 1/2^{n+1} \end{cases}$$

[5] We construct the kernel of the linear function $\lambda_i \mapsto \bigoplus_i \lambda_i (x_i \oplus y_i)$

Let N be the number of observed values, and X_i^c represents the indicator that the i^{th} value equals c (following a Bernoulli distribution), so that the counter corresponding to c is $X^c = \sum_{i=1}^{N} X_i^c$. Now we have to discriminate between the distributions of X^c with $c \neq T$, and the distribution of X^T:

$$\begin{aligned}
\Pr(X_i^T = 1) = \Pr(x = T) &= (1 - 1/2^{n+1})/2^n + 1/2^{n+1} = (3/2 - 1/2^{n+1})/2^n \\
&\implies \mathbf{E}[X^T] = N(3/2 - 1/2^{n+1})/2^n \\
\Pr(X_i^c = 1) = \Pr(x = c) &= (1 - 1/2^{n+1})/2^n \\
&\implies \mathbf{E}[X^c] = N(1 - 1/2^{n+1})/2^n \\
&\implies \mathbf{E}[X^T] \geq 3/2 \cdot \mathbf{E}[X^c]
\end{aligned}$$

We use the Chernoff bound to get a lower bound on the probability that a given counter is higher than the average value of X^T:

$$\Pr(X^c \geq \mathbf{E}[X^T]) \leq Pr(X^c \geq 3/2 \cdot \mathbf{E}[X^c]) \leq e^{-N(1-1/2^{n+1})/2^{n+1}}$$

and assuming the counters are independent:

$$\begin{aligned}
Pr(X^c < \mathbf{E}[X^T]) &\geq 1 - e^{-N(1-1/2^{n+1})/2^{n+1}} \\
Pr(\forall c \neq T : X^c < \mathbf{E}[X^T]) &\geq (1 - e^{-N(1-1/2^{n+1})/2^{n+1}})^{2^n}
\end{aligned}$$

This expression will asymptotically converge to a strictly positive constant when $e^{-N(1-1/2^{n+1})/2^{n+1}} \simeq 2^{-n}$. Therefore, we use

$$N \simeq n \ln(2) \cdot \frac{2^{n+1}}{(1 - 1/2^{n+1})} = \mathcal{O}(n \cdot 2^n).$$

Since we observe 2 values per quadruples, this makes $\mathcal{O}(n \cdot 2^n)$ quadruples. Moreover, the event '$X^T \geq \mathbf{E}[X^T]$' has a probability close to 0.5, therefore after $\mathcal{O}(n \cdot 2^n)$ quadruples, we indeed have a $\Omega(1)$ probability that X^T is greater than all of the other counters, which allows to recover the value T. Performing the attack until the end with probability $\Omega(1)$ also requires $\mathcal{O}(n \cdot 2^n)$ quadruples.

To get to this result some assumptions have been made, like the independence of the counters, but they all tend to be either conservative or asymptotically true.

5.3 Attacking `1kf9`

`1kf9` is a single-key variant of `3kf9` suggested in [8], and later withdrawn. Since the structure of `1kf9` is the same as the structure of `3kf9`, we can use the same attack. However, in the next section, we give an attack with birthday complexity using properties of the `fix` functions.

6 Attacks using collision in `fix` functions

Finally, we show attacks against single key variant of beyond-birthday-bound MACs based on `fix` functions, as defined by by Datta, Dutta, Nandi, Paul and

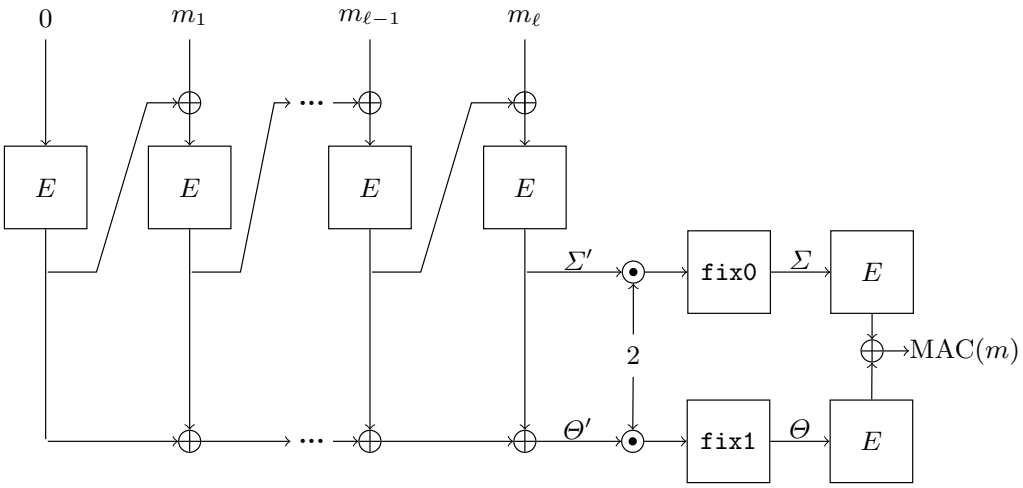


Fig. 7. Diagram for `1kf9` with a $\ell-$block message.

Zhang [8,9]. The `fix` functions just fix the least significant bit an n-bit value to zero or one, and are used for domain separation:

$$\texttt{fix0} : x \mapsto x_{[1:n]}|0 \qquad\qquad \texttt{fix1} : x \mapsto x_{[1:n]}|1$$

Datta *et al.* used those function to build a single-key variant of `PMAC+` called `1kPMAC+` [9], and a single-key variant of `3kf9` called `1kf9` [8], both with security up to $2^{2n/3}$ queries. However, `1kf9` has been withdrawn because of issues in its security proof. In this section, we exploit trivial collisions in the `fix` functions to build colliding pairs or quadruples more easily:

$$\texttt{fix0}(x) = \texttt{fix0}(x \oplus 1) \qquad\qquad \texttt{fix1}(x) = \texttt{fix1}(x \oplus 1)$$

This allows a more straightforward attack against `1kPMAC+` with the same complexity as the attacks in Section 4, and an attack against `1kf9` [8] with birthday complexity, violating its security claims.

6.1 Attacking `1kf9`

The `1kf9` mode uses the `fix` function for domain separation to build a single-key variant of `3kf9`, as shown in Figure 7:

$$\begin{aligned} \sigma_0 &= 0 & \sigma_i &= E(\sigma_{i-1} \oplus m_i) \\ \Sigma'(M) &= \sigma_\ell & \Sigma(M) &= 2 \odot \texttt{fix0}(\Sigma'(M)) \\ \Theta'(M) &= \textstyle\bigoplus_{i=1}^{\ell} \sigma_i & \Theta(M) &= 2 \odot \texttt{fix1}(\Theta'(M)) \\ \mathrm{MAC}(M) &= E(\Sigma(M)) \oplus E(\Theta(M)) \end{aligned}$$

Attack. Because of a mistake in the proof of `1kf9`, we can use pairs of messages instead of quadruples. More precisely, instead of looking for a quadruple with pairwise collisions in Σ and Θ, we look for a pair of message X, Y colliding on Σ', and with a difference in Θ' that will be absorbed by the `fix1` function. Therefore, we define the relation $\mathcal{R}$ as:

$$\mathcal{R}(X,Y) := \begin{cases} \Sigma'(X) = \Sigma'(Y) \\ 2\Theta'(X) = 2\Theta'(Y) \oplus 1 \end{cases} \Rightarrow \mathrm{MAC}(X) = \mathrm{MAC}(Y).$$

We build the messages with different postfixes, parametrized by u:

$$X = \phi_u(x) = x \parallel u \qquad\qquad Y = \psi_u(y) = y \parallel u \oplus d,$$

where d is the inverse of 2 in the finite field. With this construction, we have

$$\begin{aligned}
\Sigma'(\phi_u(x)) &= E\big(u \oplus E(x \oplus E(0))\big) \\
\Theta'(\phi_u(x)) &= E\big(u \oplus E(x \oplus E(0))\big) \oplus E\big(x \oplus E(0)\big) \oplus E\big(0\big) \\
\Sigma'(\psi_u(y)) &= E\big(u \oplus d \oplus E(y \oplus E(0))\big) \\
\Theta'(\psi_u(y)) &= E\big(u \oplus d \oplus E(y \oplus E(0))\big) \oplus E\big(y \oplus E(0)\big) \oplus E\big(0\big)
\end{aligned}$$

In particular, we observe

$$\begin{aligned}
E(x \oplus E(0)) \oplus E(y \oplus E(0)) = d \Leftrightarrow \Sigma'(\phi_u(x)) &= \Sigma'(\psi_u(y)) \\
\Rightarrow \Theta'(\phi_u(x)) \oplus \Theta'(\psi_u(y)) &= d \\
\Rightarrow \mathrm{MAC}(\phi_u(x)) &= \mathrm{MAC}(\psi_u(y)). \qquad (8)
\end{aligned}$$

From this observation, we construct a birthday attack against `1kf9`. We build two lists:

$$L_0 = \Big\{\mathrm{MAC}(\phi_0(x)) \ : \ x < 2^{n/2}\Big\} \qquad L_1 = \Big\{\mathrm{MAC}(\psi_0(y)) \ : \ y < 2^{n/2}\Big\},$$

and we look for a match between the lists. We expect on average one pair to match randomly, and one pair to match because of (8). Moreover, when we have a collision candidate $L_0[x], L_1[y]$, we can verify whether it is a right pair by comparing $\mathrm{MAC}(x \parallel 1)$ and $\mathrm{MAC}(y \parallel d \oplus 1)$.

Therefore, we find a pair satisfying $\mathcal{R}(X,Y)$ with complexity $2^{n/2}$, and this leads to simple forgeries using (8). This contradicts the security proof of `1kf9` given in[8]. Note that this attack is still valid if we use different multiplications for the two branches in the finalization function.

6.2 Attacking `1kPMAC+`

The `1kPMAC+` mode uses the `fix` function for domain separation to build a single-key variant of `PMAC+`, as shown in Figure 8.

$$\begin{aligned}
\Sigma'(M) &= \textstyle\bigoplus_{i=1}^{\ell} \tilde{E}_i(m_i) & \Sigma(M) &= \texttt{fix0}(\Sigma'(M)) \\
\Theta'(M) &= \textstyle\bigoplus_{i=1}^{\ell} 2^{\ell+1-i} \odot \tilde{E}_i(m_i) & \Theta(M) &= \texttt{fix1}(\Theta'(M)) \\
\mathrm{MAC}(M) &= E(\Sigma(M)) \oplus E(\Theta(M))
\end{aligned}$$

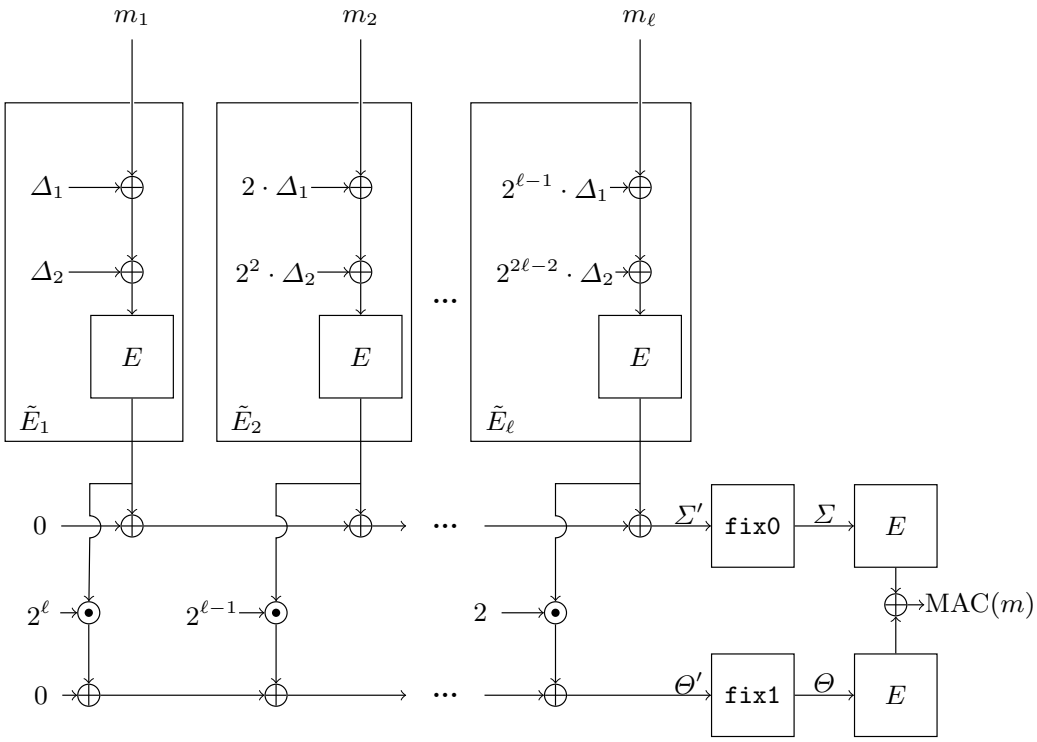


Fig. 8. Diagram for `1kPMAC+` with a ℓ-block message where $\Delta_1 = E(1)$ and $\Delta_2 = E(2)$.

Attack. Since the `fix` functions used in the finalization have collisions, we can build a variant of the attacks from Section 4 using differences in Σ' and/or Θ' that are absorbed by the `fix` functions. More precisely, we use the following relation $\mathcal{R}$ on quadruple of messages:

$$\mathcal{R}(X,Y,Z,T) := \begin{cases} \Sigma'(X) = \Sigma(Y)' \oplus 1 \\ \Theta'(Y) = \Theta(Z)' \oplus 1 \\ \Sigma'(Z) = \Sigma(T)' \oplus 1 \\ \Theta'(T) = \Theta(X)' \oplus 1 \end{cases}$$
$$\Rightarrow \mathrm{MAC}(X) \oplus \mathrm{MAC}(Y) \oplus \mathrm{MAC}(Z) \oplus \mathrm{MAC}(T) = 0.$$

We can find quadruple of messages satisfying $\mathcal{R}$ using a single message injection function:

$$\phi_u(i) = u \,\|\, i$$
$$X = \phi_u(x) = u \,\|\, x \quad Y = \psi_u(y) = u \,\|\, y \quad Z = \phi_u(z) = u \,\|\, z \quad T = \psi_u(t) = u \,\|\, t$$

Indeed we have

$$\mathrm{MAC}(\phi_u(i)) = E\Big(\texttt{fix0}\big(\underbrace{\tilde{E}_1(u) \oplus \tilde{E}_2(x)}_{\Sigma'_u(i)}\big)\Big) \oplus E\Big(\texttt{fix1}\big(\underbrace{4\tilde{E}_1(u) \oplus 2\tilde{E}_2(x)}_{\Theta'_u(i)}\big)\Big)$$

We observe that:

$$\mathcal{R}(x,y,z,t) \Leftrightarrow \begin{cases} \tilde{E}_2(x) = \tilde{E}_2(y) \oplus 1 \\ \tilde{E}_2(z) = \tilde{E}_2(t) \oplus 1 \\ 2\tilde{E}_2(x) = 2\tilde{E}_2(z) \oplus 1 \\ 2\tilde{E}_2(y) = 2\tilde{E}_2(t) \oplus 1 \end{cases}$$

$$\Leftrightarrow \begin{cases} \tilde{E}_2(x) \oplus \tilde{E}_2(y) \oplus \tilde{E}_2(z) \oplus \tilde{E}_2(t) = 0 \\ \tilde{E}_2(x) = \tilde{E}_2(y) \oplus 1 \\ \tilde{E}_2(x) = \tilde{E}_2(z) \oplus d \end{cases}$$

Therefore, $\mathcal{R}$ defines a $3n-$bit relation that is independent of the value u. This can be used for attacks in the same way as in the previous sections, using a single list

$$L = \left\{ \mathrm{MAC}(\phi_0(x)) \,\|\, \mathrm{MAC}(\phi_1(x)) \,\|\, \mathrm{MAC}(\phi_2(x)) \ : \ x < 2^{3n/4} \right\}$$

We can find a quadruple of four distinct values (x,y,z,t) such that $L[x] \oplus L[y] \oplus L[z] \oplus L[t] = 0$ with $\tilde{\mathcal{O}}(2^{3n/2})$ operations, using a memory of size $\mathcal{O}(2^{3n/4})$, and this easily leads to forgeries.

7 Conclusion

In this paper we have introduced a cryptanalysis technique to attack double-block-hash MACs using quadruples of messages. We show three variants of the technique, with attacks with $\mathcal{O}(2^{3n/4})$ queries against `SUM-ECBC`, `GCM-SIV2`, `PMAC+`, `LightMAC+`, `1kPMAC+` and `3kf9`. All these modes have a security proof up to $2^{2n/3}$ queries, but no attacks with fewer than 2^n queries were known before our work.

Our main attacks are in the information theoretic model, and an attacker would need more than 2^n operations to perform a forgery. On the other hand, we also have a variant of the attack against `SUM-ECBC` and `GCM-SIV2` with time complexity $\tilde{O}(2^{6n/7})$. This opens the path for attack with total complexity below 2^n for other double-block-hash MACs.

We believe that studying generic attacks is important in order to understand the security of these MACs, and is needed in addition to security proofs. In particular our results show that they do not reach full security, and we invalidate a recent proof for `LightMAC+`. However, there is still a gap between the $2^{2n/3}$ bound of the proofs, and our attacks with $\mathcal{O}(2^{3n/4})$ queries. Further work is needed to determine whether the attacks can be improved, or whether better proofs are possible.

Acknowledgement

Mridul Nandi is supported by R.C.Bose Centre for Cryptology and Security. Part of this work was supported by the French DGA.

A SageMath Implementation

In order to verify that the algorithm is correct, we have implemented the attack against SUM-ECBC with complexity $\tilde{\mathcal{O}}(2^{6n/7})$ given in Section 3.1 with SageMath:

```
xor  = lambda x, y: x.__xor__(y)
txor = lambda a,b: tuple(xor(u,v) for u,v in zip(a,b) )
def random_perm(n):
  pp = Permutations(n).random_element()
  return lambda x: pp(x+1)-1

def CBC(E,M):
  x = 0
  for m in M:
    x = E(x.__xor__(m))
  return x
def SUMECBC(E1,E2,E3,E4,M):
  a = E2(CBC(E1,M))
  b = E4(CBC(E3,M))
  return a.__xor__(b)

E1, E2, E3, E4 = (random_perm(2^21) for _ in range(4))
MAC = lambda x: SUMECBC(E1,E2,E3,E4,x)
print "Values to recover       | {0:06x} {1:06x}".format(
  xor(E1(0),E1(1)), xor(E3(0),E3(1)))

print "Generating data..."
L1,L2,L3,L4 = [], [], [], []
for i in range(2^12):
  if (i&0b000000000111 == 0): L1.append(i)
  if (i&0b000000111000 == 0): L2.append(i)
  if (i&0b000111000000 == 0): L3.append(i)
  if (i&0b111000000000 == 0): L4.append(i)
def macs(u,i):
  x = (0,0)
  for j in range(i,2^21,2^12):
    x = txor(x,(MAC([u,j]), MAC([u, xor(1,j)])))
  return (i,x)
L1 = [ macs(0,i) for i in L1 ]
L2 = [ macs(0,i) for i in L2 ]
L3 = [ macs(1,i) for i in L3 ]
L4 = [ macs(1,i) for i in L4 ]

print "Looking for quadruples..."
L13 = sorted((txor(a[1],b[1]),a[0],b[0]) for a in L1 for b in L3)
L24 = sorted((txor(a[1],b[1]),a[0],b[0]) for a in L2 for b in L4)
```

```
i,j = 0,0
while i<len(L13) and j<len(L24):
  if L13[i][0] == L24[j][0]:
    if L13[i] != L24[j]:
      print "{:06x} {:06x} {:06x} {:06x} | {:06x} {:06x}".format(
        L13[i][1], L13[i][2], L24[j][1], L24[j][2],
        xor(L13[i][1],L13[i][2]), xor(L13[i][1],L24[j][2]))
    if L13[i] < L24[j]:
      i+=1
    else:
      j+=1
  elif L13[i][0] < L24[j][0]:
    i+=1
  else:
    j+=1
```

References

1. An, J.H., Bellare, M.: Constructing VIL-MACs from FIL-MACs: Message authentication under weakened assumptions. In: Wiener, M.J. (ed.) CRYPTO'99. LNCS, vol. 1666, pp. 252–269. Springer, Heidelberg (Aug 1999)
2. Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: Koblitz, N. (ed.) CRYPTO'96. LNCS, vol. 1109, pp. 1–15. Springer, Heidelberg (Aug 1996)
3. Bellare, M., Guérin, R., Rogaway, P.: XOR MACs: New methods for message authentication using finite pseudorandom functions. In: Coppersmith, D. (ed.) CRYPTO'95. LNCS, vol. 963, pp. 15–28. Springer, Heidelberg (Aug 1995)
4. Bellare, M., Kilian, J., Rogaway, P.: The security of the cipher block chaining message authentication code. Journal of Computer and System Sciences 61(3), 362–399 (2000)
5. Black, J., Rogaway, P.: A block-cipher mode of operation for parallelizable message authentication. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 384–397. Springer, Heidelberg (Apr / May 2002)
6. Chose, P., Joux, A., Mitton, M.: Fast correlation attacks: An algorithmic point of view. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 209–221. Springer, Heidelberg (Apr / May 2002)
7. Cogliati, B., Seurin, Y.: EWCDM: An efficient, beyond-birthday secure, nonce-misuse resistant MAC. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016, Part I. LNCS, vol. 9814, pp. 121–149. Springer, Heidelberg (Aug 2016)
8. Datta, N., Dutta, A., Nandi, M., Paul, G., Zhang, L.: Building single-key beyond birthday bound message authentication code. Cryptology ePrint Archive, Report 2015/958 (2015), http://eprint.iacr.org/2015/958
9. Datta, N., Dutta, A., Nandi, M., Paul, G., Zhang, L.: Single key variant of PMAC_Plus. IACR Trans. Symm. Cryptol. 2017(4), 268–305 (2017)
10. Dinur, I., Leurent, G.: Improved generic attacks against hash-based MACs and HAIFA. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part I. LNCS, vol. 8616, pp. 149–168. Springer, Heidelberg (Aug 2014)

11. Dutta, A., Jha, A., Nandi, M.: Tight security analysis of EHtM MAC. IACR Trans. Symm. Cryptol. 2017(3), 130–150 (2017)
12. Ferguson, N.: Authentication weaknesses in GCM. Comment to NIST (2005), http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/CWC-GCM/Ferguson2.pdf
13. Computer data authentication. National Bureau of Standards, NIST FIPS PUB 113, U.S. Department of Commerce (1985)
14. Fuhr, T., Leurent, G., Suder, V.: Collision attacks against CAESAR candidates - forgery and key-recovery against AEZ and Marble. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015, Part II. LNCS, vol. 9453, pp. 510–532. Springer, Heidelberg (Nov / Dec 2015)
15. Gilbert, E.N., MacWilliams, F.J., Sloane, N.J.: Codes which detect deception. Bell Labs Technical Journal 53(3), 405–424 (1974)
16. Guo, J., Peyrin, T., Sasaki, Y., Wang, L.: Updates on generic attacks against HMAC and NMAC. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part I. LNCS, vol. 8616, pp. 131–148. Springer, Heidelberg (Aug 2014)
17. Iwata, T.: New blockcipher modes of operation with beyond the birthday bound security. In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 310–327. Springer, Heidelberg (Mar 2006)
18. Iwata, T., Kurosawa, K.: OMAC: One-key CBC MAC. In: Johansson, T. (ed.) FSE 2003. LNCS, vol. 2887, pp. 129–153. Springer, Heidelberg (Feb 2003)
19. Iwata, T., Mennink, B., Vizár, D.: CENC is optimally secure. Cryptology ePrint Archive, Report 2016/1087 (2016), http://eprint.iacr.org/2016/1087
20. Iwata, T., Minematsu, K.: Stronger security variants of GCM-SIV. IACR Trans. Symm. Cryptol. 2016(1), 134–157 (2016), http://tosc.iacr.org/index.php/ToSC/article/view/539
21. Iwata, T., Minematsu, K., Peyrin, T., Seurin, Y.: ZMAC: A fast tweakable block cipher mode for highly secure message authentication. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part III. LNCS, vol. 10403, pp. 34–65. Springer, Heidelberg (Aug 2017)
22. Jaulmes, É., Joux, A., Valette, F.: On the security of randomized CBC-MAC beyond the birthday paradox limit: A new construction. In: Daemen, J., Rijmen, V. (eds.) FSE 2002. LNCS, vol. 2365, pp. 237–251. Springer, Heidelberg (Feb 2002)
23. Joux, A.: Multicollisions in iterated hash functions. Application to cascaded constructions. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 306–316. Springer, Heidelberg (Aug 2004)
24. Knudsen, L.R., Mitchell, C.J.: Analysis of 3gpp-mac and two-key 3gpp-mac. Discrete Applied Mathematics 128(1), 181 – 191 (2003), http://www.sciencedirect.com/science/article/pii/S0166218X02004444, international Workshop on Coding and Cryptography (WCC2001).
25. Lee, C., Kim, J., Sung, J., Hong, S., Lee, S.: Forgery and key recovery attacks on PMAC and mitchell's TMAC variant. In: Batten, L.M., Safavi-Naini, R. (eds.) ACISP 06. LNCS, vol. 4058, pp. 421–431. Springer, Heidelberg (Jul 2006)
26. Leurent, G., Peyrin, T., Wang, L.: New generic attacks against hash-based MACs. In: Sako, K., Sarkar, P. (eds.) ASIACRYPT 2013, Part II. LNCS, vol. 8270, pp. 1–20. Springer, Heidelberg (Dec 2013)
27. List, E., Nandi, M.: Revisiting full-PRF-secure PMAC and using it for beyond-birthday authenticated encryption. In: Handschuh, H. (ed.) CT-RSA 2017. LNCS, vol. 10159, pp. 258–274. Springer, Heidelberg (Feb 2017)
28. List, E., Nandi, M.: ZMAC$^+$ – an efficient variable-output-length variant of ZMAC. IACR Trans. Symm. Cryptol. 2017(4), 306–325 (2017)

29. Luykx, A., Preneel, B., Tischhauser, E., Yasuda, K.: A MAC mode for lightweight block ciphers. In: Peyrin, T. (ed.) FSE 2016. LNCS, vol. 9783, pp. 43–59. Springer, Heidelberg (Mar 2016)
30. Mennink, B., Neves, S.: Encrypted davies-meyer and its dual: Towards optimal security using mirror theory. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part III. LNCS, vol. 10403, pp. 556–583. Springer, Heidelberg (Aug 2017)
31. Minematsu, K.: How to thwart birthday attacks against MACs via small randomness. In: Hong, S., Iwata, T. (eds.) FSE 2010. LNCS, vol. 6147, pp. 230–249. Springer, Heidelberg (Feb 2010)
32. Naito, Y.: Full PRF-secure message authentication code based on tweakable block cipher. In: Au, M.H., Miyaji, A. (eds.) ProvSec 2015. LNCS, vol. 9451, pp. 167–182. Springer, Heidelberg (Nov 2015)
33. Naito, Y.: Blockcipher-based MACs: Beyond the birthday bound without message length. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017, Part III. LNCS, vol. 10626, pp. 446–470. Springer, Heidelberg (Dec 2017)
34. Naito, Y.: Improved security bound of LightMAC_Plus and its single-key variant. In: Smart, N.P. (ed.) CT-RSA 2018. LNCS, vol. 10808, pp. 300–318. Springer, Heidelberg (Apr 2018)
35. Nikolic, I., Sasaki, Y.: Refinements of the k-tree algorithm for the generalized birthday problem. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015, Part II. LNCS, vol. 9453, pp. 683–703. Springer, Heidelberg (Nov / Dec 2015)
36. Peyrin, T., Wang, L.: Generic universal forgery attack on iterative hash-based MACs. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 147–164. Springer, Heidelberg (May 2014)
37. Preneel, B., van Oorschot, P.C.: MDx-MAC and building fast MACs from hash functions. In: Coppersmith, D. (ed.) CRYPTO'95. LNCS, vol. 963, pp. 1–14. Springer, Heidelberg (Aug 1995)
38. Preneel, B., van Oorschot, P.C.: On the security of two MAC algorithms. In: Maurer, U.M. (ed.) EUROCRYPT'96. LNCS, vol. 1070, pp. 19–32. Springer, Heidelberg (May 1996)
39. Rogaway, P.: Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In: Lee, P.J. (ed.) ASIACRYPT 2004. LNCS, vol. 3329, pp. 16–31. Springer, Heidelberg (Dec 2004)
40. Wagner, D.: A generalized birthday problem. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 288–303. Springer, Heidelberg (Aug 2002)
41. Wegman, M.N., Carter, L.: New hash functions and their use in authentication and set equality. Journal of Computer and System Sciences 22, 265–279 (1981)
42. Yasuda, K.: The sum of CBC MACs is a secure PRF. In: Pieprzyk, J. (ed.) CT-RSA 2010. LNCS, vol. 5985, pp. 366–381. Springer, Heidelberg (Mar 2010)
43. Yasuda, K.: A new variant of PMAC: Beyond the birthday bound. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 596–609. Springer, Heidelberg (Aug 2011)
44. Zhang, L., Wu, W., Sui, H., Wang, P.: 3kf9: Enhancing 3GPP-MAC beyond the birthday bound. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 296–312. Springer, Heidelberg (Dec 2012)

Breaking Symmetric Cryptosystems using Quantum Period Finding

Marc Kaplan[1,2], Gaëtan Leurent[3] Anthony Leverrier[3], and María Naya-Plasencia[3]

[1] LTCI, Télécom ParisTech, 23 avenue d'Italie, 75214 Paris CEDEX 13, France
[2] School of Informatics, University of Edinburgh,
10 Crichton Street, Edinburgh EH8 9AB, UK
[3] Inria Paris, France

Abstract. Due to Shor's algorithm, quantum computers are a severe threat for public key cryptography. This motivated the cryptographic community to search for quantum-safe solutions. On the other hand, the impact of quantum computing on secret key cryptography is much less understood. In this paper, we consider attacks where an adversary can query an oracle implementing a cryptographic primitive in a quantum superposition of different states. This model gives a lot of power to the adversary, but recent results show that it is nonetheless possible to build secure cryptosystems in it.

We study applications of a quantum procedure called *Simon's algorithm* (the simplest quantum period finding algorithm) in order to attack symmetric cryptosystems in this model. Following previous works in this direction, we show that several classical attacks based on finding collisions can be dramatically sped up using Simon's algorithm: finding a collision requires $\Omega(2^{n/2})$ queries in the classical setting, but when collisions happen with some hidden periodicity, they can be found with only $O(n)$ queries in the quantum model.

We obtain attacks with very strong implications. First, we show that the most widely used modes of operation for authentication and authenticated encryption (*e.g.* CBC-MAC, PMAC, GMAC, GCM, and OCB) are completely broken in this security model. Our attacks are also applicable to many CAESAR candidates: CLOC, AEZ, COPA, OTR, POET, OMD, and Minalpher. This is quite surprising compared to the situation with encryption modes: Anand *et al.* show that standard modes are secure with a quantum-secure PRF.

Second, we show that Simon's algorithm can also be applied to slide attacks, leading to an exponential speed-up of a classical symmetric cryptanalysis technique in the quantum model.

Keywords: post-quantum cryptography, symmetric cryptography, quantum attacks, block ciphers, modes of operation, slide attack.

1 Introduction

The goal of post-quantum cryptography is to prepare cryptographic primitives to resist quantum adversaries, *i.e.* adversaries with access to a quantum computer.

Indeed, cryptography would be particularly affected by the development of large-scale quantum computers. While currently used asymmetric cryptographic primitives would suffer from devastating attacks due to Shor's algorithm [42], the status of symmetric ones is not so clear: generic attacks, which define the security of ideal symmetric primitives, would get a quadratic speed-up thanks to Grover's algorithm [23], hinting that doubling the key length could restore an equivalent ideal security in the post-quantum world. Even though the community seems to consider the issue settled with this solution [6], only very little is known about real world attacks, that determine the real security of used primitives. Very recently, this direction has started to draw attention, and interesting results have been obtained. New theoretical frameworks to take into account quantum adversaries have been developed [11,12,19,22,15,2].

Simon's algorithm [43] is central in quantum algorithm theory. Historically, it was an important milestone in the discovery by Shor of his celebrated quantum algorithm to solve integer factorization in polynomial time [42]. Interestingly, Simon's algorithm has also been applied in the context of symmetric cryptography. It was first used to break the 3-round Feistel construction [30] and then to prove that the Even-Mansour construction [31] is insecure with superposition queries. While Simon's problem (which is the problem solved with Simon's algorithm) might seem artificial at first sight, it appears in certain constructions in symmetric cryptography, in which ciphers and modes typically involve a lot of structure.

These first results, although quite striking, are not sufficient for evaluating the security of actual ciphers. Indeed, the confidence we have on symmetric ciphers depends on the amount of cryptanalysis that was performed on the primitive. Only this effort allows researchers to define the security margin which measures how far the construction is from being broken. Thanks to the large and always updated cryptanalysis toolbox built over the years in the *classical* world, we have solid evaluations of the security of the primitives against classical adversaries. This is, however, no longer the case in the post-quantum world, *i.e.* when considering quantum adversaries.

We therefore need to build a complete cryptanalysis toolbox for quantum adversaries, similar to what has been done for the classical world. This is a fundamental step in order to correctly evaluate the post-quantum security of current ciphers and to design new secure ciphers for the post-quantum world.

Our results. We make progresses in this direction, and open new surprising and important ranges of applications for Simon's algorithm in symmetric cryptography:

1. The original formulation of Simon's algorithm is for functions whose collisions happen only at some hidden period. We extend it to functions that have more collisions. This leads to a better analysis of previous applications of Simon's algorithm in symmetric cryptography.
2. We then show an attack against the LRW construction, used to turn a block-cipher into a tweakable block cipher [32]. Like the results on 3-round Feistel and Even-Mansour, this is an example of construction with provable security in the classical setting that becomes insecure against a quantum adversary.

3. Next, we study block cipher modes of operation. We show that some of the most common modes for message authentication and authenticated encryption are completely broken in this setting. We describe forgery attacks against standardized modes (CBC-MAC, PMAC, GMAC, GCM, and OCB), and against several CAESAR candidates, with complexity only $O(n)$, where n is the size of the block. In particular, this partially answers an open question by Boneh and Zhandry [13]: "Do the CBC-MAC or NMAC constructions give quantum-secure PRFs?".

   Those results are in stark contrast with a recent analysis of encryption modes in the same setting: Anand *et al.* show that some classical encryption modes are secure against a quantum adversary when using a quantum-secure PRF [3]. Our results imply that some authentication and authenticated encryption schemes remain insecure with *any* block cipher.
4. The last application is a quantization of slide attacks, a popular family of cryptanalysis that is independent of the number of rounds of the attacked cipher. Our result is the first exponential speed-up obtained directly by a quantization of a classical cryptanalysis technique, with complexity dropping from $O(2^{n/2})$ to $O(n)$, where n is the size of the block.

These results imply that for the symmetric primitives we analyze, doubling the key length is not sufficient to restore security against quantum adversaries. A significant effort on quantum cryptanalysis of symmetric primitives is thus crucial for our long-term trust in these cryptosystems.

The attack model. We consider attacks against classical cryptosystems using quantum resources. This general setting broadly defines the field of post-quantum cryptography. But attacking specific cryptosystems requires a more precise definition of the operations the adversary is allowed to perform. The simplest setting allows the adversary to perform local quantum computation. For instance, this can be modeled by the quantum random oracle model, in which the adversary can query the oracle in an arbitrary superposition of the inputs [11,14,48,44]. A more practical setting allows quantum queries to the hash function used to instantiate the oracle on a quantum computer.

We consider here a much stronger model in which, in addition to local quantum operations, an adversary is granted an access to a possibly remote cryptographic oracle in superposition of the inputs, and obtains the corresponding superposition of outputs. In more detail, if the encryption oracle is described by a classical function $\mathcal{O}_k : \{0,1\}^n \to \{0,1\}^n$, then the adversary can make standard quantum queries $|x\rangle|y\rangle \mapsto |x\rangle|\mathcal{O}_k(x) \oplus y\rangle$, where x and y are arbitrary n-bit strings and $|x\rangle$, $|y\rangle$ are the corresponding n-qubit states expressed in the computational basis. A circuit representing the oracle is given in Figure 1. Moreover, any superposition $\sum_{x,y} \lambda_{x,y}|x\rangle|y\rangle$ is a valid input to the quantum oracle, who then returns $\sum_{x,y} \lambda_{x,y}|x\rangle|y \oplus O_k(x)\rangle$. In previous works, these attacks have been called *superposition attacks* [19], *quantum chosen message attacks* [13] or *quantum security* [47].

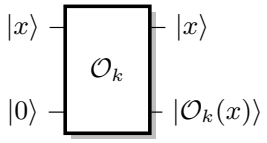


Fig. 1. The quantum cryptographic oracle.

Simon's algorithm requires the preparation of the uniform superposition of all n-bit strings, $\frac{1}{\sqrt{2^n}} \sum_x |x\rangle|0\rangle$[4]. For this input, the quantum encryption oracle returns $\frac{1}{\sqrt{2^n}} \sum_x |x\rangle|\mathcal{O}_k(x)\rangle$, the superposition of all possible pairs of plaintext-ciphertext. It might seem at first that this model gives an overwhelming power to the adversary and is therefore uninteresting. Note, however, that the laws of quantum mechanics imply that the measurement of such a $2n$-qubit state can only reveal $2n$ bits of information, making this model nontrivial.

The simplicity of this model, together with the fact that it encompasses any reasonable model of quantum attacks makes it very interesting. For instance, [12] gave constructions of message authenticated codes that remain secure against superposition attacks. A similar approach was initiated by [19], who showed how to construct secure multiparty protocols when an adversary can corrupt the parties in superposition. A protocol that is proven secure in this model may truthfully be used in a quantum world.

Our work shows that superposition attacks, although they are not trivial, allow new powerful strategies for the adversary. Modes of operation that are provably secure against classical attacks can then be broken. There exist a few options to prevent the attacks that we present here. A possibility is to forbid all kind of quantum access to a cryptographic oracle. In a world where quantum resources become available, this restriction requires a careful attention. This can be achieved for example by performing a quantum measurement of any incoming quantum query to the oracle. But this task involves meticulous engineering of quantum devices whose outcome remains uncertain. Even information theoretically secure quantum cryptography remains vulnerable to attacks on their implementations, as shown by attacks on quantum key distribution [49,34,45].

A more realistic approach is to develop a set of protocols that remains secure against superposition attacks. Another advantage of this approach is that it also covers more advanced scenarios, for example when an encryption device is given to the adversary as an obfuscated algorithm. Our work shows how important it is to develop protocols that remain secure against superposition attacks.

Regarding symmetric cryptanalysis, we have already mentioned the protocol of Boneh and Zhandry for MACs that remains secure against superposition attacks. In particular, we answer negatively to their question asking wether CBC-MAC is secure in their model. Generic quantum attacks against symmetric cryptosystems have also been considered. For instance, [27] studies the security of iterated block ciphers, and Anand et al. investigated the security of various

[4] When there is no ambiguity, we write $|0\rangle$ for the state $|0 \dots 0\rangle$ of appropriate length.

modes of operations for encryption against superposition attacks [3]. They show that OFB and CTR remain secure, while CBC and CFB are not secure in general (with attacks involving Simon's algorithm), but are secure if the underlying PRF is quantum secure. Recently, [28] considers symmetric families of cryptanalysis, describing quantum versions of differential and linear attacks.

Cryptographic notions like indistinguishability or semantic security are well understood in a classical world. However, they become difficult to formalize when considering quantum adversaries. The quantum chosen message model is a good framework to study these [22,15,2].

In this paper, we consider forgery attacks: the goal of the attacker is to forge a tag for some arbitrary message, without the knowledge of the secret key. In a quantum setting, we follow the EUF-qCMA security definition that was given by Boneh and Zhandry [12]. A message authentication code is broken by a quantum existential forgery attack if after q queries to the cryptographic oracle, the adversary can generate at least $q + 1$ valid messages with corresponding tags.

Organization. The paper is organized as follows. First, Section 2 introduces Simon's algorithm and explains how to modify it in order to handle functions that only approximately satisfy Simon's promise. This variant seems more appropriate for symmetric cryptography and may be of independent interest. Section 3 summarizes known quantum attacks against various constructions in symmetric cryptography. Section 4 presents the attack against the LRW constructions. In Section 5, we show how Simon's algorithm can be used to obtain devastating attacks on several widely used modes of operations: CBC-MAC, PMAC, GMAC, GCM, OCB, as well as several CAESAR candidates. Section 6 shows the application of the algorithm to slide attacks, providing an exponential speed-up. The paper ends in Section 7 with a conclusion, pointing out possible new directions and applications.

2 Simon's algorithm and attack strategy

In this section, we present Simon's problem [43] and the quantum algorithm for efficiently solving it. The simplest version of our attacks directly exploits this algorithm in order to recover some secret value of the encryption algorithm. Previous works have already considered such attacks against 3-round Feistel schemes and the Even-Mansour construction (see Section 3 for details).

Unfortunately, it is not always possible to recast an attack in terms of Simon's problem. More precisely, Simon's problem is a promise problem, and in many cases, the relevant promise (that only a structured class of collisions can occur) is not satisfied, far from it in fact. We show in Theorem 1 below that, however, these additional collisions do not lead to a significant increase of the complexity of our attacks.

2.1 Simon's problem and algorithm

We first describe Simon's problem, and then the quantum algorithm for solving it. We refer the reader to the recent review by Montanaro and de Wolf on quantum

property testing for various applications of this algorithm [37]. We assume here a basic knowledge of the quantum circuit model. We denote the addition and multiplication in a field with 2^n elements by "$\oplus$" and "$\cdot$", respectively.

We consider that the access to the input of Simon's problem, a function f, is made by querying it. A classical query oracle is a function $x \mapsto f(x)$. To run Simon's algorithm, it is required that the function f can be queried quantum-mechanically. More precisely, it is supposed that the algorithm can make arbitrary quantum superpositions of queries of the form $|x\rangle|0\rangle \mapsto |x\rangle|f(x)\rangle$.

Simon's problem is the following:

Simon's problem: Given a Boolean function $f : \{0,1\}^n \to \{0,1\}^n$ and the promise that there exists $s \in \{0,1\}^n$ such that for any $(x, y) \in \{0,1\}^n$, $[f(x) = f(y)] \Leftrightarrow [x \oplus y \in \{0^n, s\}]$, the goal is to find s.

This problem can be solved classically by searching for collisions. The optimal time to solve it is therefore $\Theta(2^{n/2})$. On the other hand, Simon's algorithm solves this problem with quantum complexity $O(n)$. Recall that the Hadamard transform $H^{\otimes n}$ applied on an n-qubit state $|x\rangle$ for some $x \in \{0,1\}^n$ gives $H^{\otimes n}|x\rangle = \frac{1}{\sqrt{2^n}} \sum_{y\in\{0,1\}^n} (-1)^{x\cdot y}|y\rangle$, where $x \cdot y := x_1 y_1 \oplus \cdots \oplus x_n y_n$.

The algorithm repeats the following five quantum steps.

1. Starting with a $2n$-qubit state $|0\rangle|0\rangle$, one applies a Hadamard transform $H^{\otimes n}$ to the first register to obtain the quantum superposition
$$\frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} |x\rangle|0\rangle.$$
2. A quantum query to the function f maps this to the state
$$\frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} |x\rangle|f(x)\rangle.$$
3. Measuring the second register in the computational basis yields a value $f(z)$ and collapses the first register to the state:
$$\frac{1}{\sqrt{2}}(|z\rangle + |z \oplus s\rangle).$$
4. Applying again the Hadamard transform $H^{\otimes n}$ to the first register gives:
$$\frac{1}{\sqrt{2}} \frac{1}{\sqrt{2^n}} \sum_{y\in\{0,1\}^n} (-1)^{y\cdot z} \left(1 + (-1)^{y\cdot s}\right) |y\rangle.$$
5. The vectors y such that $y \cdot s = 1$ have amplitude 0. Therefore, measuring the state in the computational basis yields a random vector y such that $y \cdot s = 0$.

By repeating this subroutine $O(n)$ times, one obtains $n - 1$ independent vectors orthogonal to s with high probability, and s can be recovered using basic linear algebra. Theorem 1 gives the trade-off between the number of repetitions of the subroutine and the success probability of the algorithm.

2.2 Dealing with unwanted collisions

In our cryptanalysis scenario, it is not always the case that the promise of Simon's problem is perfectly satisfied. More precisely, by construction, there will always exist an s such that $f(x) = f(x \oplus s)$ for any input x, but there might be many more collisions than those of this form. If the number of such unwanted collisions is too large, one might not be able to obtain a full rank linear system of equations from Simon's subroutine after $O(n)$ queries. Theorem 1 rules this out provided that f does not have too many collisions of the form $f(x) = f(x \oplus t)$ for some $t \notin \{0, s\}$.

For $f : \{0,1\}^n \to \{0,1\}^n$ such that $f(x \oplus s) = f(x)$ for all x, consider

$$\varepsilon(f, s) = \max_{t \in \{0,1\}^n \setminus \{0,s\}} \Pr_x[f(x) = f(x \oplus t)]. \tag{1}$$

This parameter quantifies how far the function is from satisfying Simon's promise. For a random function, one expects $\varepsilon(f, s) = \Theta(n2^{-n})$, following the analysis of [18]. On the other hand, for a constant function, $\varepsilon(f, s) = 1$ and it is impossible to recover s.

The following theorem, whose proof can be found in Appendix A, shows the effect of unwanted collisions on the success probability of Simon's algorithm.

Theorem 1 (Simon's algorithm with approximate promise). *If $\varepsilon(f, s) \leq p_0 < 1$, then Simon's algorithm returns s with cn queries, with probability at least $1 - \left(2\left(\frac{1+p_0}{2}\right)^c\right)^n$.*

In particular, choosing $c \geq 3/(1 - p_0)$ ensures that the error decreases exponentially with n. To apply our results, it is therefore sufficient to prove that $\varepsilon(f, s)$ is bounded away from 1.

Finally, if we apply Simon's algorithm without any bound on $\varepsilon(f, s)$, we can not always recover s unambiguously. Still if we select a random value t orthogonal to all vectors u_i returned by each step of the algorithm, t satisfy $f(x \oplus t) = f(x)$ with high probability.

Theorem 2 (Simon's algorithm without promise). *After cn steps of Simon's algorithm, if t is orthogonal to all vectors u_i returned by each step of the algorithm, then $\Pr_x[f(x \oplus t) = f(t)] \geq p_0$ with probability at least $1 - \left(2\left(\frac{1+p_0}{2}\right)^c\right)^n$.*

In particular, choosing $c \geq 3/(1 - p_0)$ ensures that the probability is exponentially close to 1.

2.3 Attack strategy

The general strategy behind our attacks exploiting Simon's algorithm is to start with the encryption oracle $E_k : \{0,1\}^n \to \{0,1\}^n$ and exhibit a new function f that satisfies Simon's promise with two additional properties: the adversary should be able to query f in superposition if he has quantum oracle access to E_k, and the knowledge of the string s should be sufficient to break the cryptographic scheme. In the following, this function is called Simon's function.

In most cases, our attacks correspond to a classical collision attack. In particular, the value s will usually be the difference in the internal state after processing a fixed pair of messages (α_0, α_1), *i.e.* $s = E(\alpha_0) \oplus E(\alpha_1)$. The input of f will be inserted into the state with the difference s so that $f(x) = f(x \oplus s)$.

In our work, this function f is of the form:

$$f^1 : x \quad \mapsto P(\widetilde{E}(x) + \widetilde{E}(x \oplus s)) \quad \text{or,}$$
$$f^2 : b, x \quad \mapsto \begin{cases} \widetilde{E}(x) & \text{if } b = 0, \\ \widetilde{E}(x \oplus s) & \text{if } b = 1, \end{cases}$$

where $\widetilde{E}$ is a simple function obtained from E_k and P a permutation. It is immediate to see that f^1 and f^2 have periods s for f^1 or $1||s$ for f^2.

In most applications, Simon's function satisfies $f(x) = f(y)$ for $y \oplus x \in \{0, s\}$, but also for additional inputs x, y. Theorem 1 extends Simon's algorithm precisely to this case. In particular, if the additional collisions of f are random, then Simon's algorithm is successful. When considering explicit constructions, we can not in general prove that the unwanted collisions *are* random, but rather that they *look random enough*. In practice, if the function $\varepsilon(f, s)$ is not bounded, then some of the primitives used in the construction have are far from ideal. We can show that this happens with low probability, and would imply an classical attack against the system. Applying Theorem 1 is not trivial, but it stretches the range of application of Simon's algorithm far beyond its original version.

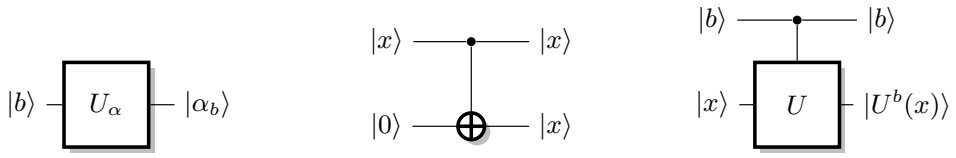
Construction of Simon's functions. To make our attacks as clear as possible, we provide the diagrams of circuits computing the function f. These circuits use a little number of basic building blocks represented in Figure 2.

In our attacks, we often use a pair of arbitrary constants α_0 and α_1. The choice of the constant is indexed by a bit b. We denote by U_α the gate that maps b to α_b (See Figure 2.1). For simplicity, we ignore here the additional qubits required in practice to make the transform reversible through padding.

Although it is well known that arbitrary quantum states cannot be cloned, we use the $CNOT$ gate to copy classical information. More precisely, a CNOT gate can copy states in the computational basis: $CNOT : |x\rangle|0\rangle \to |x\rangle|x\rangle$. This transform is represented in Figure 2.2.

Finally, any unitary transform U can be controlled by a bit b. This operation, denoted U^b maps x to $U(x)$ if $b = 1$ and leaves x unchanged otherwise. In the quantum setting, the qubit $|b\rangle$ can be in a superposition of 0 and 1, resulting in a superposition of $|x\rangle$ and $|U(x)\rangle$. The attacks that we present in the following sections only make use of this procedure when the attacker knows a classical description of the unitary to be controlled. In particular, we do not apply it to the cryptographic oracle.

When computing Simon's function, *i.e.* the function f on which Simon's algorithm is applied, the registers containing the value of f must be unentangled with any other working register. Otherwise, these registers, which might hinder the periodicity of the function, have to be taken into account in Simon's algorithm and the whole procedure could fail.



2.1. One-to-one mapping. **2.2.** CNOT gate. **2.3.** Controlled Unitary.

Fig. 2. Circuit representation of basic building blocks.

3 Previous works

Previous works have used Simon's algorithm to break the security of classical constructions in symmetric cryptography: the Even-Mansour construction and the 3-round Feistel scheme. We now explain how these attacks work with our terminology and extend two of the results. First, we show that the attack on the Feistel scheme can be extended to work with random functions, where the original analysis held only for random permutations. Second, using our analysis Simon's algorithm with approximate promise, we make the number of queries required to attack the Even-Mansour construction more precise. These observations have been independently made by Santoli and Schaffner [40]. They use a slightly different approach, which consists in analyzing the run of Simon's algorithm for these specific cases.

3.1 Applications to a three-round Feistel scheme

The Feistel scheme is a classical construction to build a random permutation out of random functions or random permutations. In a seminal work, Luby and Rackoff proved that a three-round Feistel scheme is a secure pseudo-random permutation [33].

A three-round Feistel scheme with input (x_L, x_R) and output $(y_L, y_R) = E(x_L, x_R)$ is built from three round functions R_1, R_2, R_3 as (see Figure 3):

$$(u_0, v_0) = (x_L, x_R), \quad (u_i, v_i) = (v_{i-1} \oplus R_i(u_{i-1}), u_{i-1}), \quad (y_L, y_R) = (u_3, v_3).$$

In order to distinguish a Feistel scheme from a random permutation in a quantum setting, Kuwakado and Morii [30] consider the case were the R_i are permutations, and define the following function, with two arbitrary constants α_0 and α_1 such that $\alpha_0 \neq \alpha_1$:

$$\begin{aligned} f : \{0,1\} \times \{0,1\}^n &\to \{0,1\}^n \\ b, x &\mapsto y_R \oplus \alpha_b, \quad \text{where } (y_R, y_L) = E(\alpha_b, x) \\ f(b,x) &= R_2(x \oplus R_1(\alpha_b)) \end{aligned}$$

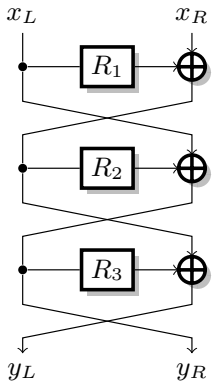


Fig. 3. Three-round Feistel scheme.

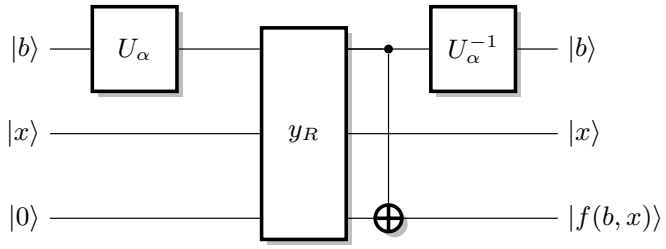


Fig. 4. Simon's function for Feistel.

In particular, this f satisfies $f(b,x) = f(b \oplus 1, x \oplus R_1(\alpha_0) \oplus R_1(\alpha_1))$. Moreover,

$$f(b',x') = f(b,x) \Leftrightarrow x' \oplus R_1(\alpha_{b'}) = x \oplus R_1(\alpha_b)$$
$$\Leftrightarrow \begin{cases} x' \oplus x = 0 & \text{if } b' = b \\ x' \oplus x = R_1(\alpha_0) \oplus R_1(\alpha_1) & \text{if } b' \neq b \end{cases}$$

Therefore, the function satisfies Simon's promise with $s = 1 \parallel R_1(\alpha_0) \oplus R_1(\alpha_1)$, and we can recover $R_1(\alpha_0) \oplus R_1(\alpha_1)$ using Simon's algorithm. This gives a distinguisher, because Simon's algorithm applied to a random permutation returns zero with high probability. This can be seen from Theorem 2, using the fact that with overwhelming probability[18], there is no value $t \neq 0$ such that $\Pr_x[f(x \oplus t) = f(x)] > 1/2$ for a random permutation f.

We can also verify that the value $R_1(\alpha_0) \oplus R_1(\alpha_1)$ is correct with two additional classical queries $(y_L, y_R) = E(\alpha_0, x)$ and $(y'_L, y'_R) = E(\alpha_1, x \oplus R_1(\alpha_0) \oplus R_1(\alpha_1))$ for a random x. If the value is correct, we have $y_R \oplus y'_R = \alpha_0 \oplus \alpha_1$.

Note that in their attack, Kuwakado and Morii implicitly assume that the adversary can query in superposition an oracle that returns solely the left part y_L of the encryption. If the adversary only has access to the complete encryption oracle E, then a query in superposition would return two *entangled* registers containing the left and right parts, respectively. In principle, Simon's algorithm requires the register containing the input value to be completely disentangled from the others.

Feistel scheme with random functions. Kuwakado and Morii [30] analyze only the case where the round functions R_i are permutations. We now extend this analysis to *random functions* R_i. The function f defined above still satisfies $f(b,x) = f(b \oplus 1, x \oplus R_1(\alpha_0) \oplus R_1(\alpha_1))$, but it doesn't satisfy the exact promise of Simon's algorithm: there are additional collisions in f, between inputs with random differences. However, the previous distinguisher is still valid: at the end of Simon's algorithm, there exist at least one non-zero value orthogonal to all

the values y measured at each step: s. This would not be the case with a random permutation.

Moreover, we can show that $\varepsilon(f, 1\|s) < 1/2$ with overwhelming probability, so that Simon's algorithm still recovers $1 \| s$ following Theorem 1. If $\varepsilon(f, 1 \| s) > 1/2$, there exists (τ, t) with $(\tau, t) \notin \{(0,0), (1,s)\}$ such that: $\Pr[f(b,x) = f(b \oplus \tau, x \oplus t)] > 1/2$. Assume first that $\tau = 0$, this implies:

$$\Pr[f(0,x) = f(0, x \oplus t)] > 1/2 \quad \text{or} \quad \Pr[f(1,x) = f(1, x \oplus t)] > 1/2.$$

Therefore, for some b, $\Pr[R_2(x \oplus R_1(\alpha_b)) = R_2(x \oplus t \oplus R_1(\alpha_b))] > 1/2$, *i.e.* $\Pr[R_2(x) = R_2(x \oplus t)] > 1/2$. Similarly, if $\tau = 1$, $\Pr[R_2(x \oplus R_1(\alpha_0)) = R_2(x \oplus t \oplus R_1(\alpha_1))] > 1/2$, *i.e.* $\Pr[R_2(x) = R_2(x \oplus t \oplus R_1(\alpha_0) \oplus R_1(\alpha_1))] > 1/2$.

To summarize, if $\varepsilon(f, 1 \| s) > 1/2$, there exists $u \neq 0$ such that $\Pr[R_2(x) = R_2(x \oplus u)] > 1/2$. This only happens with negligible probability for a random choice of R_2 as shown in [18].

3.2 Application to the Even-Mansour construction

The Even-Mansour construction is a simple construction to build a block cipher from a public permutation [21]. For some permutation P, the cipher is:

$$E_{k_1,k_2}(x) = P(x \oplus k_1) \oplus k_2.$$

Even and Mansour have shown that this construction is secure in the random permutation model, up to $2^{n/2}$ queries, where n is the size of the input to P.

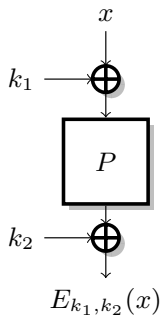


Fig. 5. Even-Mansour scheme.

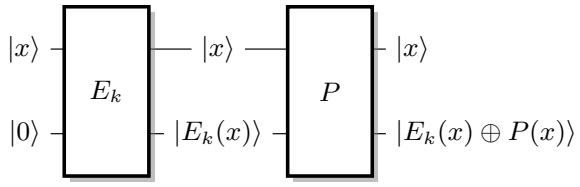


Fig. 6. Simon's function for Even-Mansour.

However, Kuwakado and Morii [31] have shown that the security of this construction collapses if an adversary can query an encryption oracle with a superposition of states. More precisely, they define the following function:

$$\begin{aligned} f : \{0,1\}^n &\to \{0,1\}^n \\ x &\mapsto E_{k_1,k_2}(x) \oplus P(x) = P(x \oplus k_1) \oplus P(x) \oplus k_2. \end{aligned}$$

In particular, f satisfies $f(x \oplus k_1) = f(x)$ (interestingly, the slide with a twist attack of Biryukov and Wagner[8] uses the same property). However, there are additional collisions in f between inputs with random differences. As in the attack against the Feistel scheme with random round functions, we use Theorem 1, to show that Simon's algorithm recovers k_1[5].

We show that $\varepsilon(f, k_1) < 1/2$ with overwhelming probability for a random permutation P, and if $\varepsilon(f, k_1) > 1/2$, then there exists a classical attack against the Even-Mansour scheme. Assume that $\varepsilon(f, k_1) > 1/2$, that is, there exists t with $t \notin \{0, k_1\}$ such that $\Pr[f(x) = f(x \oplus t)] > 1/2$, *i.e.*,

$$p = \Pr[P(x) \oplus P(x \oplus k_1) \oplus P(x \oplus t) \oplus P(x \oplus t \oplus k_1) = 0] > 1/2.$$

This correspond to higher order differential for P with probability $1/2$, which only happens with negligible probability for a random choice of P. In addition, this would imply the existence of a simple classical attack against the scheme:

1. Query $y = E_{k_1,k_2}(x)$ and $y' = E_{k_1,k_2}(x \oplus t)$
2. Then $y \oplus y' = P(x) \oplus P(x \oplus t)$ with probability at least one half

Therefore, for any instantiation of the Even-Mansour scheme with a fixed P, either there exist a classical distinguishing attack (this only happens with negligible probability with a random P), or Simon's algorithm successfully recovers k_1. In the second case, the value of k_2 can then be recovered from an additional classical query: $k_2 = E(x) \oplus P(x \oplus k_1)$.

In the next sections, we give new applications of Simon's algorithm, to break various symmetric cryptography schemes.

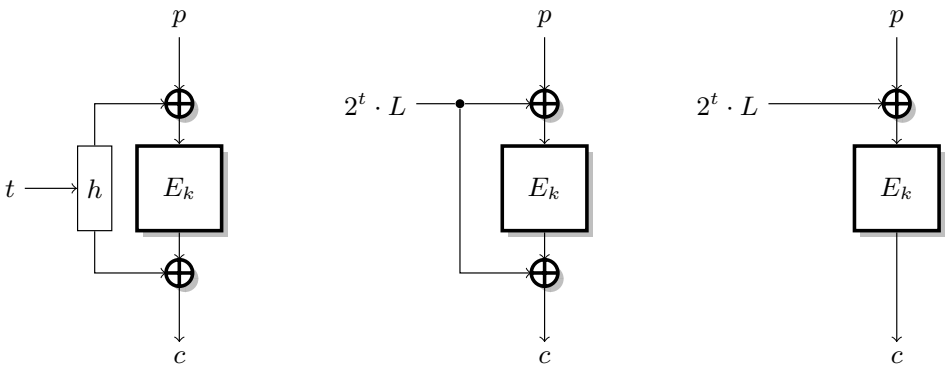
4 Application to the LRW construction

We now show a new application of Simon's algorithm to the LRW construction. The LRW construction, introduced by Liskov, Rivest and Wagner [32], turns a block cipher into a tweakable block cipher, *i.e.* a family of unrelated block ciphers. The tweakable block cipher is a very useful primitive to build modes for encryption, authentication, or authenticated encryption. In particular, tweakable block ciphers and the LRW construction were inspired by the first version of OCB, and later versions of OCB use the tweakable block ciphers formalism. The LRW construction uses a (almost) universal hash function h (which is part of the key), and is defined as (see also Figure 7):

$$\widetilde{E}_{t,k}(x) = E_k(x \oplus h(t)) \oplus h(t).$$

We now show that the LRW construction is not secure in a quantum setting. We fix two arbitrary tweaks t_0, t_1, with $t_0 \neq t_1$, and we define the following

[5] Note that Kuwakado and Morii just assume that each step of Simon's algorithm gives a random vector orthogonal to k_1. Our analysis is more formal and captures the conditions on P required for the algorithm to be successful.



7.1. LRW construction. **7.2.** XEX construction. **7.3.** XE construction.

Fig. 7. The LRW construction, and efficient instantiations XEX (CCA secure) and XE (only CPA secure).

function:

$$\begin{aligned} f : \{0,1\}^n &\to \{0,1\}^n \\ x \quad &\mapsto \widetilde{E}_{t_0,k}(x) \oplus \widetilde{E}_{t_1,k}(x) \\ f(x) &= E_k\big(x \oplus h(t_0)\big) \oplus h(t_0) \oplus E_k\big(x \oplus h(t_1)\big) \oplus h(t_1). \end{aligned}$$

Given a superposition access to an oracle for an LRW tweakable block cipher, we can build a circuit implementing this function, using the construction given in Figure 8. In the circuit, the cryptographic oracle $\widetilde{E}_{t,k}$ takes two inputs: the block x to be encrypted and the tweak t. Since the tweak comes out of $\widetilde{E}_{t,k}$ unentangled with the other register, we do not represent this output in the diagram. In practice, the output is forgotten by the attacker.

It is easy to see that this function satisfies $f(x) = f(x \oplus s)$ with $s = h(t_0) \oplus h(t_1)$. Furthermore, the quantity $\varepsilon(f, s) = \max_{t \in \{0,1\}^n \setminus \{0,s\}} \Pr[f(x) = f(x \oplus t)]$ is bounded with overwhelming probability, assuming that E_k behaves as a random permutation. Indeed if $\varepsilon(f, s) > 1/2$, there exists some t with $t \notin \{0, s\}$ such that $\Pr[f(x) = f(x \oplus t)] > 1/2$, *i.e.*,

$$\Pr[E_k\big(x\big) \oplus E_k\big(x \oplus s\big) \oplus E_k\big(x \oplus t\big)\big) \oplus E_k\big(x \oplus t \oplus s\big) = 0] > 1/2$$

This correspond to higher order differential for E_k with probability $1/2$, which only happens with negligible probability for a random permutation. Therefore, if E is a pseudo-random permutation family, $\varepsilon(f, s) \leq 1/2$ with overwhelming probability, and running Simon's algorithm with the function f returns $h(t_0) \oplus h(t_1)$. The assumption that E behaves as a PRP family is required for the security proof of LRW, so it is reasonable to make the same assumption in an attack. More concretely, a block cipher with a higher order differential with probability $1/2$ as seen above would probably be broken by classical attacks. The attack is not immediate because the differential can depend on the key, but it would seem to

indicate a structural weakness. In the following sections, some attacks can also be mounted using Theorem 2 without any assumptions on E.

In any case, there exist at least one non-zero value orthogonal to all the values y measured during Simon's algorithm: s. This would not be the case if f is a random function, which gives a distinguisher between the LRW construction and an ideal tweakable block cipher with $O(n)$ quantum queries to $\widetilde{E}$.

In practice, most instantiations of LRW use a finite field multiplication to define the universal hash function h, with a secret offset L (usually computed as $L = E_k(0)$). Two popular constructions are:

- $h(t) = \gamma(t) \cdot L$, used in OCB1 [39], OCB3 [29] and PMAC [10], with a Gray encoding γ of t,
- $h(t) = 2^t \cdot L$, the XEX construction, used in OCB2 [38].

In both cases, we can recover L from the value $h(t_0) \oplus h(t_1)$ given by the attack.

This attack is important, because many recent modes of operation are inspired by the LRW construction, and the XE and XEX instantiations, such as CAESAR candidates AEZ [24], COPA [4], OCB [29], OTR [36], Minalpher [41], OMD [17], and POET [1]. We will see in the next section that variants of this attack can be applied to each of these modes.

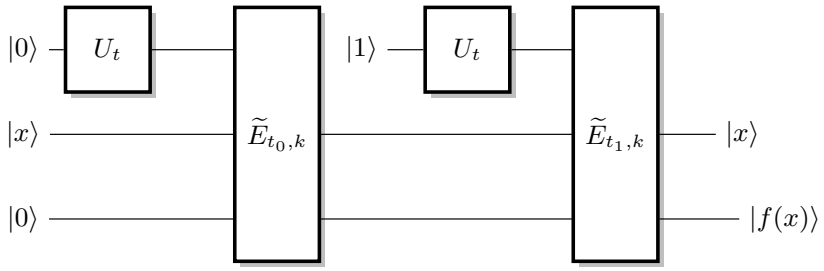


Fig. 8. Simon's function for LRW.

5 Application to block cipher modes of operations

We now give new applications of Simon's algorithm to the security of block cipher modes of operations. In particular, we show how to break the most popular and widely used block-cipher based MACs, and message authentication schemes: CBC-MAC (including variants such as XCBC [9], OMAC [25], and CMAC [20]), GMAC [35], PMAC [10], GCM [35] and OCB [29]. We also show attacks against several CAESAR candidates. In each case, the mode is proven secure up to $2^{n/2}$ in the classical setting, but we show how, by a reduction to Simon's problem, forgery attacks can be performed with superposition queries at a cost of $O(n)$.

Notations and preliminaries. We consider a block cipher E_k, acting on blocks of length n, where the subscript k denotes the key. For simplicity, we

only describe the modes with full-block messages, the attacks can trivially be extended to the more general modes with arbitrary inputs. In general, we consider a message M divided into ℓ n-bits block: $M = m_1 \| \ldots \| m_\ell$. We also assume that the MAC is not truncated, *i.e.* the output size is n bits. In most cases, the attacks can be adapted to truncated MACS.

5.1 Deterministic MACs: CBC-MAC and PMAC

We start with deterministic Message Authentication Codes, or MACs. A MAC is used to guarantee the authenticity of messages, and should be immune against forgery attacks. The standard security model is that it should be hard to forge a message with a valid tag, even given access to an oracle that computes the MAC of any chosen message (of course the forged message must not have been queried to the oracle).

To translate this security notion to the quantum setting, we assume that the adversary is given an oracle that takes a quantum superposition of messages as input, and computes the superposition of the corresponding MAC.

CBC-MAC. CBC-MAC is one of the first MAC constructions, inspired by the CBC encryption mode. Since the basic CBC-MAC is only secure when the queries are prefix-free, there are many variants of CBC-MAC to provide security for arbitrary messages. In the following we describe the Encrypted-CBC-MAC variant [5], using two keys k and k', but the attack can be easily adapted to other variants [9,25,20]. On a message $M = m_1 \| \ldots \| m_\ell$, CBC-MAC is defined as (see Figure 9):

$$x_0 = 0 \qquad x_i = E_k(x_{i-1} \oplus m_i) \qquad \text{CBC-MAC}(M) = E_{k'}(x_\ell)$$

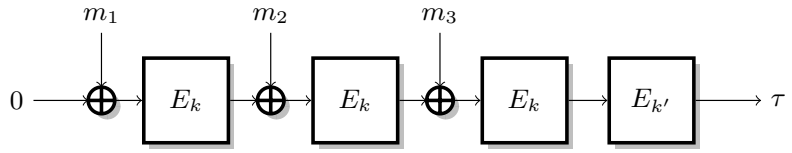


Fig. 9. Encrypt-last-block CBC-MAC.

CBC-MAC is standardized and widely used. It has been proved to be secure up to the birthday bound [5], assuming that the block cipher is indistinguishable from a random permutation.

Attack. We can build a powerful forgery attack on CBC-MAC with very low complexity using superposition queries. We fix two arbitrary message blocks α_0, α_1, with $\alpha_0 \neq \alpha_1$, and we define the following function:

$$\begin{aligned} f : \{0,1\} \times \{0,1\}^n &\to \{0,1\}^n \\ b, x \quad &\mapsto \text{CBC-MAC}(\alpha_b \| x) = E_{k'}\left(E_k\big(x \oplus E_k(\alpha_b)\big)\right). \end{aligned}$$

The function f can be computed with a single call to the cryptographic oracle, and we can build a quantum circuit for f given a black box quantum circuit for CBC-MAC$_k$. Moreover, f satisfies the promise of Simon's problem with $s = 1 \,\|\, E_k(\alpha_0) \oplus E_k(\alpha_1)$:

$$\begin{aligned} f(0,x) &= E_{k'}(E_k(x \oplus E_k(\alpha_1))), \\ f(1,x) &= E_{k'}(E_k(x \oplus E_k(\alpha_0))), \\ f(b,x) &= f(b \oplus 1, x \oplus E_k(\alpha_0) \oplus E_k(\alpha_1)). \end{aligned}$$

More precisely:

$$\begin{aligned} f(b',x') = f(b,x) &\Leftrightarrow x \oplus E_k(\alpha_b) = x' \oplus E_k(\alpha_{b'}) \\ &\Leftrightarrow \begin{cases} x' \oplus x = 0 & \text{if } b' = b \\ x' \oplus x = E_k(\alpha_0) \oplus E_k(\alpha_1) & \text{if } b' \neq b \end{cases} \end{aligned}$$

Therefore, an application of Simon's algorithm returns $E_k(\alpha_0) \oplus E_k(\alpha_1)$. This allows to forge messages easily:

1. Query the tag of $\alpha_0 \,\|\, m_1$ for an arbitrary block m_1;
2. The same tag is valid for $\alpha_1 \,\|\, m_1 \oplus E_k(\alpha_0) \oplus E_k(\alpha_1)$.

In order to break the formal notion of EUF-qCMA security, we must produce $q+1$ valid tags with only q queries to the oracle. Let $q' = O(n)$ denote the number of of quantum queries made to learn $E_k(\alpha_0) \oplus E_k(\alpha_1)$. The attacker will repeats the forgery step step $q'+1$ times, in order to produce $2(q'+1)$ messages with valid tags, after a total of $2q'+1$ classical and quantum queries to the cryptographic oracle. Therefore, CBC-MAC is broken by a quantum existential forgery attack.

After some exchange at early stages of the work, an extension of this forgery attack has been found by Santoli and Schaffner [40]. Its main advantage is to handle oracles that accept input of fixed length, while our attack works for oracles accepting messages of variable length.

PMAC. PMAC is a parallelizable block-cipher based MAC designed by Rogway [38]. PMAC is based on the XE construction: the construction uses secret offsets Δ_i derived from the secret key to turn the block cipher into a tweakable block cipher. More precisely, the PMAC algorithm is defined as

$$c_i = E_k(m_i \oplus \Delta_i) \qquad \text{PMAC}(M) = E_k^*\big(m_\ell \oplus \sum c_i\big)$$

where E^* is a tweaked variant of E. We omit the generation of the secret offsets because they are irrelevant to our attack.

First attack. When PMAC is used with two-block messages, it has the same structure as CBC-MAC: $\text{PMAC}(m_1 \,\|\, m_2) = E_k^*(m_2 \oplus E_k(m_1 \oplus \Delta_0))$. Therefore we can use the attack of the previous section to recover $E_k(\alpha_0) \oplus E_k(\alpha_1)$ for arbitrary values of α_0 and α_1. Again, this leads to a simple forgery attack. First, query the tag of $\alpha_0 \,\|\, m_1 \,\|\, m_2$ for arbitrary blocks m_1, m_2. The same tag is

valid for $\alpha_1 \parallel m_1 \parallel m_2 \oplus E_k(\alpha_0) \oplus E_k(\alpha_1)$. As for CBC-MAC, these two steps can be repeated $t+1$ times, where t is the number of quantum queries issued. The adversary then produces $2(t+1)$ messages after only $2t+1$ queries to the cryptographic oracle.

Second attack. We can also build another forgery attack on PMAC where we recover the difference between two offsets Δ_i, following the attack against LRW given in Section 4. More precisely, we use the following function:

$$\begin{aligned} f : \{0,1\}^n &\to \{0,1\}^n \\ m &\mapsto \mathrm{PMAC}(m \parallel m \parallel 0^n) = E_k^* \left(E_k(m \oplus \Delta_0) \oplus E_k(m \oplus \Delta_1)\right). \end{aligned}$$

In particular, it satisfies $f(m \oplus s) = f(m)$ with $s = \Delta_0 \oplus \Delta_1$. Furthermore, we can show that $\varepsilon(f, s) \leq 1/2$ when E is a good block cipher[6], and we can apply Simon's algorithm to recover $\Delta_0 \oplus \Delta_1$. This allows to create forgeries as follows:

1. Query the tag of $m_1 \parallel m_1$ for an arbitrary block m_1;
2. The same tag is valid for $m_1 \oplus \Delta_0 \oplus \Delta_1 \parallel m_1 \oplus \Delta_0 \oplus \Delta_1$.

As mentioned in Section 4, the offsets in PMAC are defined as $\Delta_i = \gamma(i) \cdot L$, with $L = E_k(0)$ and γ a Gray encoding. This allows to recover L from $\Delta_0 \oplus \Delta_1$, as $L = (\Delta_0 \oplus \Delta_1) \cdot (\gamma(0) \oplus \gamma(1))^{-1}$. Then we can compute all the values Δ_i, and forge arbitrary messages.

We can also mount an attack without any assumption on $\varepsilon(f, s)$, using Theorem 2. Indeed, with a proper choice of parameters, Simon's algorithm will return a value $t \neq 0$ that satisfies $\Pr_x[f(x \oplus t) = f(x)] \geq 1/2$. This value is not necessarily equal to s, but it can also be used to create forgeries in the same way, with success probability at least $1/2$.

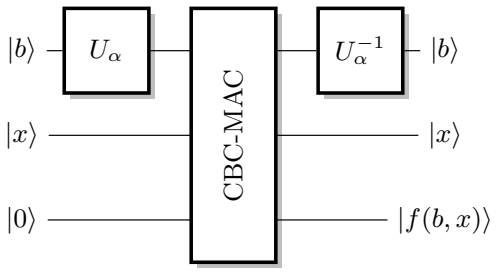


Fig. 10. Simon's function for CBC-MAC.

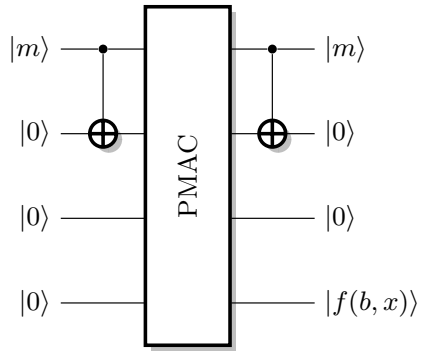


Fig. 11. Simon's function for the second attack against PMAC.

[6] Since this attack is just a special case of the LRW attack of Section 4, we don't repeat the detailed proof.

5.2 Randomized MAC: GMAC

GMAC is the underlying MAC of the widely used GCM standard, designed by McGrew and Viega [35], and standardized by NIST. GMAC follows the Carter-Wegman construction [16]: it is built from a universal hash function, using polynomial evaluation in a Galois field. As opposed to the constructions of the previous sections, GMAC is a randomized MAC; it requires a second input N, which must be non-repeating (a nonce). GMAC is essentially defined as:

$$\begin{aligned}\mathrm{GMAC}(N, M) &= \mathrm{GHASH}(M \parallel \mathrm{len}(M)) \oplus E_k(N||1)\\ \mathrm{GHASH}(M) &= \sum_{i=1}^{\mathrm{len}(M)} m_i \cdot H^{\mathrm{len}(M)-i+1} \quad \text{with } H = E_k(0),\end{aligned}$$

where $\mathrm{len}(M)$ is the length of M.

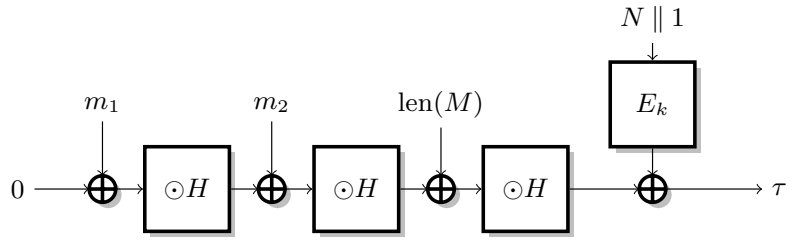


Fig. 12. GMAC

Attack. When the polynomial is evaluated with Horner's rule, the structure of GMAC is similar to that of CBC-MAC (see Figure 12). For a two-block message, we have $\mathrm{GMAC}(m_1 \parallel m_2) = \big((m_1 \cdot H) \oplus m_2\big) \cdot H \oplus E_k(N \parallel 1)$. Therefore, we us the same f as in the CBC-MAC attack, with fixed blocks α_0 and α_1:

$$\begin{aligned}f_N : \{0,1\} \times \{0,1\}^n &\to \{0,1\}^n\\ b, x \quad &\mapsto \mathrm{GMAC}(N, \alpha_b \parallel x) = \alpha_b \cdot H^2 \oplus x \cdot H \oplus E_k(N||1).\end{aligned}$$

In particular, we have:

$$\begin{aligned}f(b', x') = f(b, x) &\Leftrightarrow \alpha_b \cdot H^2 \oplus x \cdot H = \alpha_{b'} \cdot H^2 \oplus x' \cdot H\\ &\Leftrightarrow \begin{cases} x' \oplus x = 0 & \text{if } b' = b\\ x' \oplus x = (\alpha_0 \oplus \alpha_1) \cdot H & \text{if } b' \neq b \end{cases}\end{aligned}$$

Therefore f_N satisfies the promise of Simon's algorithm with $s = 1 \parallel (\alpha_0 \oplus \alpha_1) \cdot H$.

Role of the nonce. There is an important caveat regarding the use of the nonce. In a classical setting, the nonce is chosen by the adversary under the constraint that it is non-repeating, *i.e.* the oracle computes $N, M \mapsto \mathrm{GMAC}(N, M)$. However, in the quantum setting, we don't have a clear definition of non-repeating

if the nonce can be in superposition. To sidestep the issue, we use a weaker security notion where the nonce is chosen at random by the oracle, rather than by the adversary (following the IND-qCPA definition of [13]). The oracle is then $M \mapsto (r, \mathrm{GMAC}(r, M))$. If we can break the scheme in this model, the attack will also be valid with any reasonable CPA security definition.

In this setting we can access the function f_N only for a random value of N. In particular, we cannot apply Simon's algorithm as is, because this requires $O(n)$ queries to the *same* function f_N. However, a single step of Simon's algorithm requires a single query to the f_N function, and returns a vector orthogonal to s, for any random choice of N. Therefore, we can recover $(\alpha_0 \oplus \alpha_1) \cdot H$ after $O(n)$ steps, even if each step uses a different value of N. Then, we can recover H easily, and it is easy to generate forgeries when H is known:

1. Query the tag of $N, m_1 \| m_2$ for arbitrary blocks m_1, m_2 (under a random nonce N).
2. The same tag is valid for $m_1 \oplus 1 \| m_2 \oplus H$ (with the same nonce N).

As for CBC-MAC, repeating these two steps leads to an existential forgery attack.

5.3 Classical Authenticated Encryption Schemes: GCM and OCB

We now give applications of Simon's algorithm to break the security of standardized authenticated encryption modes. The attacks are similar to the attacks against authentication modes, but these authenticated encryption modes are nonce-based. Therefore we have to pay special attention to the nonce, as in the attack against GMAC. In the following, we assume that the nonce is randomly chosen by the MAC oracle, in order to avoid issues with the definition of non-repeating nonce in a quantum setting.

Extending MAC attacks to authenticated encryption schemes. We first present a generic way to apply MAC attacks in the context of an authenticated encryption scheme. More precisely, we assume that the tag of the authenticated encryption scheme is computed as $f(g(A), h(M, N))$, *i.e.* the authentication of the associated data A is independent of the nonce N. This is the case in many practical schemes (*e.g.* GCM, OCB) for efficiency reasons.

In this setting, we can use a technique similar to our attack against GMAC: we define a function $M \mapsto f_N(M)$ for a fixed nonce N, such that for any nonce N, $f_N(M) = f_N(M \oplus \Delta)$ for some secret value Δ. Next we use Simon's algorithm to recover Δ, where each step of Simon's algorithm is run with a random nonce, and returns a vector orthogonal to Δ. Finally, we can recover Δ, and if f_N was carefully built, the knowledge of Δ is sufficient for a forgery attack.

The CCM mode is a notable exception, where all the computations depend on the nonce. In particular, there is no obvious way to apply our attacks to CCM.

Extending GMAC attack to GCM. GCM is one of the most widely used authenticated encryption modes, designed by McGrew and Viega [35]. GMAC is the composition of the counter mode for encryption with GMAC (computed over the associated data and the ciphertext) for authentication.

In particular, when the message is empty, GCM is just GMAC, and we can use the attack of the previous section to recover the hash key H. This immediately allows a forgery attack.

OCB. OCB is another popular authenticated encryption mode, with a very high efficiency, designed by Rogaway *et al.* [39,38,29]. Indeed, OCB requires only ℓ block cipher calls to process an ℓ-block message, while GCM requires ℓ block cipher calls, and ℓ finite field operations. OCB is build from the LRW construction discussed in Section 4. OCB takes as input a nonce N, a message $M = m_1 \| \dots \| m_\ell$, and associated data $A = a_1 \| \dots a_@$, and returns a ciphertext $C = c_1 \| \dots \| c_\ell$ and a tag τ:

$$c_i = E_k(m_i \oplus \Delta_i^N) \oplus \Delta_i^N, \quad \tau = E_k\left(\Delta_\ell'^N \oplus \sum m_i\right) \oplus \sum b_i, \quad b_i = E_k(a_i \oplus \Delta_i).$$

Extending PMAC attack to OCB. In particular, when the message is empty, OCB reduces to a randomized variant of PMAC:

$$\mathrm{OCB}_k(N, \varepsilon, A) = \phi_k(N) \oplus \sum b_i, \qquad b_i = E_k(a_i \oplus \Delta_i).$$

Note that the Δ_i values used for the associated data are independent of the nonce N. Therefore, we can apply the second PMAC attack previously given, using the following function:

$$\begin{aligned} f_N : \{0,1\}^n &\to \{0,1\}^n \\ x &\mapsto \mathrm{OCB}_k(N, \varepsilon, x \| x) \\ f_N(x) &= E_k(x \oplus \Delta_0) \oplus E_k(x \oplus \Delta_1) \oplus \phi_k(N) \end{aligned}$$

Again, this is a special case of the LRW attack of Section 4. The family of functions satisfies $f_N(a \oplus \Delta_0 \oplus \Delta_1) = f_N(a)$, for any N, and $\varepsilon(f_N, \Delta_0 \oplus \Delta_1) \leq 1/2$ with overwhelming probability if E is a PRP. Therefore we can use the variant of Simon's algorithm to recover $\Delta_0 \oplus \Delta_1$. Two messages with valid tags can then be generated by a single classical queries:

1. Query the authenticated encryption C, τ of $M, a \| a$ for an arbitrary message M, and an arbitrary block a (under a random nonce N).
2. C, τ is also a valid authenticated encryption of $M, a \oplus \Delta_0 \oplus \Delta_1 \| a \oplus \Delta_0 \oplus \Delta_1$, with the same nonce N.

Repeating these steps lead again to an existential forgery attack.

Alternative attack against OCB. For some versions of OCB, we can also mount a different attack targeting the encryption part rather than the authentication part. The goal of this attack is also to recover the secret offsets, but we target the Δ_i^N used for the encryption of the message. More precisely, we use the following function:

$$\begin{aligned} f_i : \{0,1\}^n &\to \{0,1\}^n \\ m &\mapsto c_1 \oplus c_2, \text{where } (c_1, c_2, \tau) = \mathrm{OCB}_k(N, m \| m, \varepsilon) \\ f_i(m) &= E_k(m \oplus \Delta_1^N) \oplus \Delta_1^N \oplus E_k(m \oplus \Delta_2^N) \oplus \Delta_2^N \end{aligned}$$

This function satisfies $f_N(m \oplus \Delta_1^N \oplus \Delta_2^N) = f_N(m)$ and $\varepsilon(f_N, \Delta_0^N \oplus \Delta_1^N) \leq 1/2$, with the same arguments as previously. Moreover, in OCB1 and OCB3, the offsets are derived as $\Delta_i^N = \Phi_k(N) \oplus \gamma(i) \cdot E_k(0)$ for some function Φ (based on the block cipher E_k). In particular, $\Delta_1^N \oplus \Delta_2^N$ is independent of N:

$$\Delta_1^N \oplus \Delta_2^N = (\gamma(1) \oplus \gamma(2)) \cdot E_k(0).$$

Therefore, we can apply Simon's algorithm to recover $\Delta_1^N \oplus \Delta_2^N$. Again, this leads to a forgery attack, by repeating the following two steps:

1. Query the authenticated encryption $c_1 \parallel c_2, \tau$ of $m \parallel m, A$ for an arbitrary block m, and arbitrary associated data A (under a random nonce N).
2. $c_2 \oplus \Delta_0^N \oplus \Delta_1^N \parallel c_1 \oplus \Delta_0^N \oplus \Delta_1^N, \tau$ is also a valid authenticated encryption of $m \oplus \Delta_0^N \oplus \Delta_1^N \parallel m \oplus \Delta_0^N \oplus \Delta_1^N, A$ with the same nonce N.

The forgery is valid because we swap the inputs of the first and second block ciphers. In addition, we have $\sum m_i = \sum m'_i$, so that the tag is still valid.

5.4 New Authenticated Encryption Schemes: CAESAR Candidates

In this section, we consider recent proposals for authenticated encryption, submitted to the ongoing CAESAR competition. Secret key cryptography has a long tradition of competitions: AES and SHA-3 for example, were chosen after the NIST competitions organized in 1997 and 2007, respectively. The CAESAR competition[7] aims at stimulating research on authenticated encryption schemes, and to define a portfolio of new authenticated encryption schemes. The competition is currently in the second round, with 29 remaining algorithms.

First, we point out that the attacks of the previous sections can be used to break several CAESAR candidates:

- CLOC [26] uses CBC-MAC to authenticate the message, and the associated data is processed independently of the nonce. Therefore, the CBC-MAC attack can be extended to CLOC[8].
- AEZ [24], COPA [4], OTR [36] and POET [1] use a variant of PMAC to authenticate the associated data. In both cases, the nonce is not used to process the associated data, so that we can extend the PMAC attack as we did against OCB[9].
- The authentication of associated data in OMD [17] and Minalpher [41] are also variants of PMAC (with a PRF that is not block cipher), and the attack can be applied.

In the next section, we show how to adapt the PMAC attack to Minalpher and OMD, since the primitives are different.

[7] `http://competitions.cr.yp.to/`

[8] This is not the case for the related mode SILC, because the nonce is processed before the data in CBC-MAC.

[9] Note that AEZ, COPA and POET also claim security when the nonce is misused, but our attacks are nonce-respecting.

Minalpher. Minalpher [41] is a permutation-based CAESAR candidate, where the permutation is used to build a tweakable block-cipher using the tweakable Even-Mansour construction. When the message is empty (or fixed), the authentication part of Minalpher is very similar to PMAC. With associated data $A = a_1 \| \dots a_@$, the tag is computed as:

$$b_i = P(a_i \oplus \Delta_i) \oplus \Delta_i \qquad \tau = \phi_k \left(N, M, a_@ \oplus \sum_{i=1}^{@-1} b_i \right)$$
$$\Delta_i = y^i \cdot L' \qquad L' = P(k \| 0) \oplus (k \| 0)$$

where ϕ_k is a permutation (we omit the description of ϕ_k because it is irrelevant for our attack). Since the tag is a function of $a_@ \oplus \sum_{i=1}^{@-1} b_i$, we can use the same attacks as against PMAC. For instance, we define the following function:

$$\begin{aligned} f_N : \{0,1\} \times \{0,1\}^n &\to \{0,1\}^n \\ b, x \quad &\mapsto \text{Minalpher}(N, \varepsilon, \alpha_b \| x) = \phi_k(N, \varepsilon, P(\alpha_b \oplus \Delta_1) \oplus \Delta_1 \oplus x). \end{aligned}$$

In particular, we have:

$$\begin{aligned} f_N(b', x') = f_N(b, x) &\Leftrightarrow P(\alpha_{b'} \oplus \Delta_1) \oplus x' = P(\alpha_b \oplus \Delta_1) \oplus x \\ &\Leftrightarrow \begin{cases} x' \oplus x = 0 & \text{if } b' = b \\ x' \oplus x = P(\alpha_0 \oplus \Delta_1) \oplus P(\alpha_1 \oplus \Delta_1) & \text{if } b' \neq b \end{cases} \end{aligned}$$

Since $s = P(\alpha_0 \oplus \Delta_1) \oplus P(\alpha_1 \oplus \Delta_1)$ is independent of N, we can easily apply Simon's algorithm to recover s, and generate forgeries.

OMD. OMD [17] is a compression-function-based CAESAR candidate. The internal primitive is a keyed compression function denoted F_k. Again, when the message is empty the authentication is very similar to PMAC. With associated data $A = a_1 \| \dots a_@$, the tag is computed as:

$$b_i = F_k(a_i \oplus \Delta_i) \qquad \tau = \phi_k(N, M) \oplus \sum b_i$$

We note that the Δ_i used for the associated data do not depend on the nonce. Therefore we can use the second PMAC attack with the following function:

$$\begin{aligned} f_N : \{0,1\}^n &\to \{0,1\}^n \\ x \quad &\mapsto \text{OMD}(N, \varepsilon, x \| x) \\ f_N(x) &= \phi_k(N, \varepsilon) \oplus F_k(x \oplus \Delta_1) \oplus F_k(x \oplus \Delta_2) \end{aligned}$$

This is the same form as seen when extending the PMAC attack to OCB, therefore we can apply the same attack to recover $s = \Delta_1 \oplus \Delta_2$ and generate forgeries.

6 Simon's algorithm applied to slide attacks

In this section we show how Simon's algorithm can be applied to a cryptanalysis family: slide attacks. In this case, the complexity of the attack drops again

exponentially, from $O(2^{n/2})$ to $O(n)$ and therefore becomes much more dangerous. To the best of our knowledge this is the first symmetric cryptanalytic technique that has an exponential speed-up in the post-quantum world.

The principle of slide attacks In 1999, Wagner and Biryukov introduced the technique called *slide attack* [7]. It can be applied to block ciphers made of r applications of an identical round function R, each one parametrized by the same key K. The attack works independently of the number of rounds, r. Intuitively, for the attack to work, R has to be vulnerable to known plaintext attacks.

The attacker collects $2^{n/2}$ encryptions of plaintexts. Amongst these couples of plaintext-ciphertext, with large probability, he gets a "slid" pair, that is, a pair of couples (P_0, C_0) and (P_1, C_1) such that $R(P_0) = P_1$. This immediately implies that $R(C_0) = C_1$. For the attack to work, the function R needs to allow for an efficient recognition of such pairs, which in turns makes the key extraction from R easy. A trivial application of this attack is the key-alternate cipher with blocks of n bits, identical subkeys and no round constants. The complexity is then approximately $2^{n/2}$. The speed-up over exhaustive search given by this attack is then quadratic, similar to the quantum attack based on Grover's algorithm.

This attack is successful, for example, to break the TREYFER block cipher [46], with a data complexity of 2^{32} and a time complexity of $2^{32+12} = 2^{44}$ (where 2^{12} is the cost of identifying the slid pair by performing some key guesses). Comparatively, the cost for an exhaustive search of the key is 2^{64}.

Exponential quantum speed-up of slide attacks We consider the attack represented in Figure 13. The unkeyed round function is denoted P and the whole encryption function E_k.

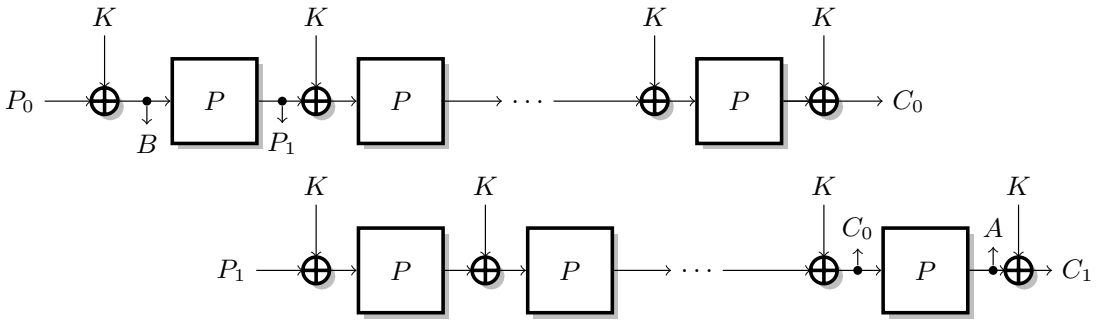


Fig. 13. Representation of a slid-pair used in a slide attack.

We define the following function:

$$f : \{0,1\} \times \{0,1\}^n \to \{0,1\}^n$$
$$b, x \quad \mapsto \begin{cases} P(E_k(x)) \oplus x & \text{if } b = 0, \\ E_k(P(x)) \oplus x & \text{if } b = 1. \end{cases}$$

The slide property shows that all x satisfy $P(E_k(x)) \oplus k = E_k(P(x \oplus k))$. This implies that f satisfies the promise of Simon's problem with $s = 1 \,\|\, k$:

$$f(0, x) = P(E_k(x)) \oplus x = E_k(P(x \oplus k)) \oplus k \oplus x = f(1, x \oplus k).$$

In order to apply Theorem 1, we bound $\varepsilon(f, 1 \,\|\, k)$, assuming that both $E_k \circ P$ and $P \circ E_k$ are indistinguishable from random permutations. If $\varepsilon(f, 1 \,\|\, k) > 1/2$, there exists (τ, t) with $(\tau, t) \notin \{(0, 0), (1, k)\}$ such that: $\Pr[f(b, x) = f(b \oplus \tau, x \oplus t)] > 1/2$. Let us assume $\tau = 0$. This implies

$$\Pr[f(0, x) = f(0, x \oplus t)] > 1/2 \quad \text{or} \quad \Pr[f(1, x) = f(1, x \oplus t)] > 1/2,$$

which is equivalent to

$$\Pr[P(E_k(x)) = P(E_k(x \oplus t)) \oplus t] > 1/2 \quad \text{or} \quad \Pr[E_k(P(x)) = E_k(P(x \oplus t)) \oplus t] > 1/2.$$

In particular, there is a differential in $P \circ E_k$ or $E_k \circ P$ with probability $1/2$. Otherwise, $\tau = 1$. This implies

$$\begin{aligned} & \Pr[P(E_k(x)) \oplus x = E_k(P(x \oplus t)) \oplus x \oplus t] > 1/2 \\ \textit{i.e.} \quad & \Pr[E_k(P(x \oplus k)) \oplus k = E_k(P(x \oplus t)) \oplus t] > 1/2. \end{aligned}$$

Again, it means there is a differential in $E_k \circ P$ with probability $1/2$.

Finally we conclude that $\varepsilon(f, 1 \,\|\, k) \leq 1/2$, unless $E_k \circ P$ or $P \circ E_k$ have differentials with probability $1/2$. If E_k behave as a random permutation, $E_k \circ P$ and $P \circ E_k$ also behave as random permutations, and these differential are only found with negligible probability. Therefore, we can apply Simon's algorithm, following Theorem 1, and recover k.

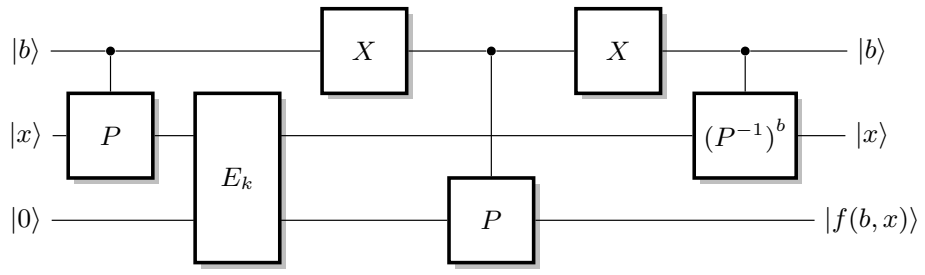


Fig. 14. Simon's function for slide attacks. The X gate is the quantum equivalent of the `NOT` gate that flips the qubit $|0\rangle$ and $|1\rangle$.

7 Conclusion

We have been able to show that symmetric cryptography is far from ready for the post quantum world. We have found exponential speed-ups on attacks on

symmetric cryptosystems. In consequence, some cryptosystems that are believed to be safe in a classical world become vulnerable in a quantum world.

With the speed-up on slide attacks, we provided the first known exponential quantum speed-up of a classical attack. This attack now becomes very powerful. An interesting follow-up would be to seek other such speed-ups of generic techniques. For authenticated encryption, we have shown that many modes of operations that are believed to be solid and secure in the classical world, become completely broken in the post-quantum world. More constructions might be broken following the same ideas.

Acknowledgements

We would like to thank Thomas Santoli and Christian Schaffner for sharing an early stage manuscript of their work [40], Michele Mosca for discussions and LTCI for hospitality. This work was supported by the Commission of the European Communities through the Horizon 2020 program under project number 645622 PQCRYPTO. MK acknowledges funding through grants ANR-12-PDOC-0022-01 and ESPRC EP/N003829/1.

References

1. Abed, F., Fluhrer, S.R., Forler, C., List, E., Lucks, S., McGrew, D.A., Wenzel, J.: Pipelineable on-line encryption. In: Cid, C., Rechberger, C. (eds.) Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers. Lecture Notes in Computer Science, vol. 8540, pp. 205–223. Springer (2014)
2. Alagic, G., Broadbent, A., Fefferman, B., Gagliardoni, T., Schaffner, C., Jules, M.S.: Computational security of quantum encryption. arXiv preprint arXiv:1602.01441 (2016)
3. Anand, M.V., Targhi, E.E., Tabia, G.N., Unruh, D.: Post-quantum security of the CBC, CFB, OFB, CTR, and XTS modes of operation. In: Takagi, T. (ed.) Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016, Fukuoka, Japan, February 24-26, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9606, pp. 44–63. Springer (2016)
4. Andreeva, E., Bogdanov, A., Luykx, A., Mennink, B., Tischhauser, E., Yasuda, K.: Parallelizable and authenticated online ciphers. In: Sako, K., Sarkar, P. (eds.) Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part I. Lecture Notes in Computer Science, vol. 8269, pp. 424–443. Springer (2013)
5. Bellare, M., Kilian, J., Rogaway, P.: The security of the cipher block chaining message authentication code. J. Comput. Syst. Sci. 61(3), 362–399 (2000)
6. Bernstein, D.J.: Introduction to post-quantum cryptography. In: Post-quantum cryptography, pp. 1–14. Springer (2009)
7. Biryukov, A., Wagner, D.: Slide attacks. In: Knudsen, L.R. (ed.) Fast Software Encryption, 6th International Workshop, FSE '99, Rome, Italy, March 24-26, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1636, pp. 245–259. Springer (1999)

8. Biryukov, A., Wagner, D.: Advanced slide attacks. In: Preneel, B. (ed.) Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding. Lecture Notes in Computer Science, vol. 1807, pp. 589–606. Springer (2000)
9. Black, J., Rogaway, P.: CBC macs for arbitrary-length messages: The three-key constructions. In: Bellare, M. (ed.) Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1880, pp. 197–215. Springer (2000)
10. Black, J., Rogaway, P.: A block-cipher mode of operation for parallelizable message authentication. In: Knudsen, L.R. (ed.) Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2332, pp. 384–397. Springer (2002)
11. Boneh, D., Dagdelen, Ö., Fischlin, M., Lehmann, A., Schaffner, C., Zhandry, M.: Random oracles in a quantum world. In: Lee, D., Wang, X. (eds.) Advances in Cryptology – ASIACRYPT 2011, Lecture Notes in Computer Science, vol. 7073, pp. 41–69. Springer Berlin Heidelberg (2011)
12. Boneh, D., Zhandry, M.: Quantum-secure message authentication codes. In: Johansson, T., Nguyen, P.Q. (eds.) Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7881, pp. 592–608. Springer (2013)
13. Boneh, D., Zhandry, M.: Secure signatures and chosen ciphertext security in a quantum computing world. In: Canetti, R., Garay, J.A. (eds.) Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II. Lecture Notes in Computer Science, vol. 8043, pp. 361–379. Springer (2013)
14. Brassard, G., Høyer, P., Kalach, K., Kaplan, M., Laplante, S., Salvail, L.: Merkle puzzles in a quantum world. In: Advances in Cryptology–CRYPTO 2011, pp. 391–410. Springer (2011)
15. Broadbent, A., Jeffery, S.: Quantum homomorphic encryption for circuits of low T-gate complexity. In: Advances in Cryptology–CRYPTO 2015, pp. 609–629. Springer (2015)
16. Carter, L., Wegman, M.N.: Universal classes of hash functions (extended abstract). In: Hopcroft, J.E., Friedman, E.P., Harrison, M.A. (eds.) Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA. pp. 106–112. ACM (1977)
17. Cogliani, S., Maimut, D., Naccache, D., do Canto, R.P., Reyhanitabar, R., Vaudenay, S., Vizár, D.: OMD: A compression function mode of operation for authenticated encryption. In: Joux, A., Youssef, A.M. (eds.) Selected Areas in Cryptography - SAC 2014 - 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8781, pp. 112–128. Springer (2014)
18. Daemen, J., Rijmen, V.: Probability distributions of correlation and differentials in block ciphers. J. Mathematical Cryptology 1(3), 221–242 (2007)
19. Damgård, I., Funder, J., Nielsen, J.B., Salvail, L.: Superposition attacks on cryptographic protocols. In: Padró, C. (ed.) Information Theoretic Security - 7th International Conference, ICITS 2013, Singapore, November 28-30, 2013, Proceedings. Lecture Notes in Computer Science, vol. 8317, pp. 142–161. Springer (2013)

20. Dworkin, M.: Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication. NIST Special Publication 800-38B, National Institute for Standards and Technology (May 2005)
21. Even, S., Mansour, Y.: A construction of a cipher from a single pseudorandom permutation. J. Cryptology 10(3), 151–162 (1997)
22. Gagliardoni, T., Hülsing, A., Schaffner, C.: Semantic security and indistinguishability in the quantum world. arXiv preprint arXiv:1504.05255 (2015)
23. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Miller, G.L. (ed.) Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996. pp. 212–219. ACM (1996)
24. Hoang, V.T., Krovetz, T., Rogaway, P.: Robust authenticated-encryption AEZ and the problem that it solves. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9056, pp. 15–44. Springer (2015)
25. Iwata, T., Kurosawa, K.: OMAC: one-key CBC MAC. In: Johansson, T. (ed.) Fast Software Encryption, 10th International Workshop, FSE 2003, Lund, Sweden, February 24-26, 2003, Revised Papers. Lecture Notes in Computer Science, vol. 2887, pp. 129–153. Springer (2003)
26. Iwata, T., Minematsu, K., Guo, J., Morioka, S.: CLOC: authenticated encryption for short input. In: Cid, C., Rechberger, C. (eds.) Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers. Lecture Notes in Computer Science, vol. 8540, pp. 149–167. Springer (2014)
27. Kaplan, M.: Quantum attacks against iterated block ciphers. CoRR abs/1410.1434 (2014)
28. Kaplan, M., Leurent, G., Leverrier, A., Naya-Plasencia, M.: Quantum differential and linear cryptanalysis. CoRR abs/1510.05836 (2015)
29. Krovetz, T., Rogaway, P.: The software performance of authenticated-encryption modes. In: Joux, A. (ed.) Fast Software Encryption - 18th International Workshop, FSE 2011, Lyngby, Denmark, February 13-16, 2011, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6733, pp. 306–327. Springer (2011)
30. Kuwakado, H., Morii, M.: Quantum distinguisher between the 3-round Feistel cipher and the random permutation. In: Information Theory Proceedings (ISIT), 2010 IEEE International Symposium on. pp. 2682–2685 (June 2010)
31. Kuwakado, H., Morii, M.: Security on the quantum-type Even-Mansour cipher. In: Information Theory and its Applications (ISITA), 2012 International Symposium on. pp. 312–316 (Oct 2012)
32. Liskov, M., Rivest, R.L., Wagner, D.: Tweakable block ciphers. J. Cryptology 24(3), 588–613 (2011)
33. Luby, M., Rackoff, C.: How to construct pseudorandom permutations from pseudorandom functions. SIAM J. Comput. 17(2), 373–386 (1988)
34. Lydersen, L., Wiechers, C., Wittmann, C., Elser, D., Skaar, J., Makarov, V.: Hacking commercial quantum cryptography systems by tailored bright illumination. Nature photonics 4(10), 686–689 (2010)
35. McGrew, D.A., Viega, J.: The security and performance of the galois/counter mode (GCM) of operation. In: Canteaut, A., Viswanathan, K. (eds.) Progress in Cryptology - INDOCRYPT 2004, 5th International Conference on Cryptology in India, Chennai, India, December 20-22, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3348, pp. 343–355. Springer (2004)

36. Minematsu, K.: Parallelizable rate-1 authenticated encryption from pseudorandom functions. In: Nguyen, P.Q., Oswald, E. (eds.) Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8441, pp. 275–292. Springer (2014)
37. Montanaro, A., de Wolf, R.: A survey of quantum property testing. arXiv preprint arXiv:1310.2035 (2013)
38. Rogaway, P.: Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In: Lee, P.J. (ed.) Advances in Cryptology - ASIACRYPT 2004, 10th International Conference on the Theory and Application of Cryptology and Information Security, Jeju Island, Korea, December 5-9, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3329, pp. 16–31. Springer (2004)
39. Rogaway, P., Bellare, M., Black, J., Krovetz, T.: OCB: a block-cipher mode of operation for efficient authenticated encryption. In: Reiter, M.K., Samarati, P. (eds.) CCS 2001, Proceedings of the 8th ACM Conference on Computer and Communications Security, Philadelphia, Pennsylvania, USA, November 6-8, 2001. pp. 196–205. ACM (2001)
40. Santoli, T., Schaffner, C.: Using simon's algorithm to attack symmetric-key cryptographic primitives. arXiv preprint arXiv:1603.07856 (2016)
41. Sasaki, Y., Todo, Y., Aoki, K., Naito, Y., Sugawara, T., Murakami, Y., Matsui, M., Hirose, S.: Minalpher v1.1. CAESAR submission (August 2015)
42. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. 26(5), 1484–1509 (1997)
43. Simon, D.R.: On the power of quantum computation. SIAM journal on computing 26(5), 1474–1483 (1997)
44. Unruh, D.: Non-interactive zero-knowledge proofs in the quantum random oracle model. In: Eurocrypt 2015. vol. 9057, pp. 755–784. Springer (2015), preprint on IACR ePrint 2014/587
45. Xu, F., Qi, B., Lo, H.K.: Experimental demonstration of phase-remapping attack in a practical quantum key distribution system. New Journal of Physics 12(11), 113026 (2010)
46. Yuval, G.: Reinventing the travois: Encryption/mac in 30 ROM bytes. In: Biham, E. (ed.) Fast Software Encryption, 4th International Workshop, FSE '97, Haifa, Israel, January 20-22, 1997, Proceedings. Lecture Notes in Computer Science, vol. 1267, pp. 205–209. Springer (1997)
47. Zhandry, M.: How to construct quantum random functions. In: 53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012. pp. 679–687. IEEE Computer Society (2012)
48. Zhandry, M.: Secure identity-based encryption in the quantum random oracle model. International Journal of Quantum Information 13(04), 1550014 (2015)
49. Zhao, Y., Fung, C.H.F., Qi, B., Chen, C., Lo, H.K.: Quantum hacking: Experimental demonstration of time-shift attack against practical quantum-key-distribution systems. Physical Review A 78(4), 042333 (2008)

A Proof of Theorem 1

The proof of Theorem 1 is based of the following lemma.

Lemma 1. *For $t \in \{0,1\}^n$, consider the function $g(x) := 2^{-n} \sum_{y \in t^\perp} (-1)^{x \cdot y}$, where $t^\perp = \{y \in \{0,1\}^n \text{ s.t. } y \cdot t = 0\}$. for any x, it satisfies*

$$g(x) = \frac{1}{2}(\delta_{x,0} + \delta_{x,t}). \tag{2}$$

Proof. If $t = 0$ then $g(x) = \sum_{y \in \{0,1\}^n} (-1)^{x \cdot y} = \delta(x, 0)$, which proves the claim. From now on, assume that $t \neq 0$. It is straightforward to check that $g(0) = g(t) = \frac{1}{2}$ because all the terms of the sum are equal to 1 and there are 2^{n-1} vectors y orthogonal to t. Since $\sum_{x \in \{0,1\}^n} g(x) = 1$, it is sufficient to prove that $g(x) \geq 0$ to establish the claim in the case $t \neq 0$. For this, decompose $g(x)$ into two terms:

$$g(x) = \sum_{y \in E_0} (-1)^{x \cdot y} - \sum_{y \in E_1} (-1)^{x \cdot y} = |E_0| - |E_1|,$$

where $E_i := \{y \in \{0,1\}^n \text{ s.t. } y \cdot x = i \text{ and } y \cdot y = 0\}$ for $i = 0, 1$. Simple counting shows that:

$$|E_0| = \begin{cases} 2^{n-1} \text{ if } x = 0, \\ 2^{n-1} \text{ if } x = t, \\ 2^{n-2} \text{ otherwise.} \end{cases}$$

In particular, $|E_0| \geq |E_1|$ which implies that $g(x) \geq 0$.

We are now ready to prove Theorem 1. Each call to the main subroutine of Simon's algorithm will return a vector u_i. If cn calls are made, one obtains cn vectors $u_1, \ldots, u_{cn}$. By construction, f is such that $f(x) = f(x \oplus s)$ and consequently, the cn vectors $u_1, \ldots, u_{cn}$ are all orthogonal to s. The algorithm is successful provided one can recover the value of s unambiguously, which is the case if the cn vectors span the $(n-1)$-dimensional space orthogonal to s. (Let us note that if the space is $(n-d)$-dimensional for some constant d, one can still recover s efficiently by testing all the vectors orthogonal to the subspace.) In other words, the failure probability p_{fail} is

$$\begin{aligned} p_{\text{fail}} &= \Pr[\dim\big(\text{Span}(u_1, \ldots, u_n)\big) \leq n-2] \\ &\leq \Pr[\exists t \in \{0,1\}^n \setminus \{0, s\} \text{ s.t. } u_1 \cdot t = u_2 \cdot t = \cdots = u_{cn} \cdot t = 0] \\ &\leq \sum_{t \in \{0,1\}^n \setminus \{0,s\}} \Pr[u_1 \cdot t = u_2 \cdot t = \cdots = u_{cn} \cdot t = 0] \\ &\leq \sum_{t \in \{0,1\}^n \setminus \{0,s\}} \big(\Pr[u_1 \cdot t = 0]\big)^{cn} \\ &\leq \max_{t \in \{0,1\}^n \setminus \{0,s\}} \big(2\Pr[u_1 \cdot t = 0]^c\big)^n \end{aligned}$$

where the second inequality results from the union bound and the third inequality follows from the fact that the results of the cn subroutines are independent.

In order to establish the theorem, it is now sufficient to show that $\Pr[u \cdot t = 0]$ is bounded away from 1 for all t, where u is the vector corresponding to the

output of Simon's subroutine. We will prove that for all $t \in \{0,1\}^n \setminus \{0, s\}$, the following inequality holds:

$$\Pr_u[u \cdot t = 0] = \frac{1}{2}\big(1 + \Pr_x[f(x) = f(x \oplus t)]\big) \leq \frac{1}{2}(1 + \varepsilon(f,s)) \leq \frac{1}{2}(1 + p_0). \quad (3)$$

In Simon's algorithm, one can wait until the last step before measuring both registers. The final state before measurement can be decomposed as:

$$\begin{aligned} 2^{-n} \sum_{x\in\{0,1\}^n} \sum_{y\in\{0,1\}^n} (-1)^{x\cdot y}|y\rangle|f(x)\rangle =& 2^{-n} \sum_{\substack{y\in\{0,1\}^n \\ \text{s.t. } y\cdot t=0}} \sum_{x\in\{0,1\}^n} (-1)^{x\cdot y}|y\rangle|f(x)\rangle \\ &+ 2^{-n} \sum_{\substack{y\in\{0,1\}^n \\ \text{s.t. } y\cdot t=1}} \sum_{x\in\{0,1\}^n} (-1)^{x\cdot y}|y\rangle|f(x)\rangle. \end{aligned}$$

The probability of obtaining u such that $u \cdot t = 0$ is given by

$$\begin{aligned} \Pr_u[u \cdot t = 0] &= \Big\| 2^{-n} \sum_{\substack{y\in\{0,1\}^n \\ \text{s.t. } y\cdot t=0}} |y\rangle \sum_{x\in\{0,1\}^n} (-1)^{x\cdot y}|f(x)\rangle \Big\|^2 \\ &= 2^{-2n} \sum_{\substack{y\in\{0,1\}^n \\ \text{s.t. } y\cdot t=0}} \sum_{x,x'\in\{0,1\}^n} (-1)^{(x\oplus x')\cdot y}\langle f(x')|f(x)\rangle \\ &= 2^{-2n} \sum_{x,x'\in\{0,1\}^n} \langle f(x')|f(x)\rangle \sum_{\substack{y\in\{0,1\}^n \\ \text{s.t. } y\cdot t=0}} (-1)^{(x\oplus x')\cdot y} \\ &= 2^{-2n} \sum_{x,x'\in\{0,1\}^n} \langle f(x')|f(x)\rangle 2^{n-1}(\delta_{x,x'} + \delta_{x',x\oplus t}) \quad (4) \\ &= 2^{-(n+1)} \left[\sum_{x\in\{0,1\}^n} \langle f(x)|f(x)\rangle + \sum_{x\in\{0,1\}^n} \langle f(x\oplus t)|f(x)\rangle \right] \quad (5) \\ &= \frac{1}{2}\left[1 + \Pr_x[f(x) = f(x \oplus t)]\right] \quad (6) \end{aligned}$$

where we used Lemma 1 proven in the appendix in Eq. 4, and $\delta_{x,x'} = 1$ if $x = x'$ and 0 otherwise.

B Proof of Theorem 2

Let t be a fixed value and $p_t = Pr_x[f(x \oplus t = f(t)]$. Following the previous analysis, the probability that the cn vectors u_i are orthogonal to t can be written as $\Pr[u_1 \cdot t = u_2 \cdot t = \cdots = u_{cn} \cdot t = 0] = \left(\frac{1+p_t}{2}\right)^{cn}$.

In particular, we can bound the probability that Simon's algorithm returns a value t with $p_t < p_0$:

$$\Pr[p_t < p_0] = \sum_{t:\, p_t<p_0} \left(\frac{1+p_t}{2}\right)^{cn} \leq 2^n \times \left(\frac{1+p_0}{2}\right)^{cn}$$

Quantum Linearization Attacks

Xavier Bonnetain[1,2], Gaëtan Leurent[2], María Naya-Plasencia[2], and André Schrottenloher[3]

[1] Institute for Quantum Computing, Department of Combinatorics and Optimization, University of Waterloo, Waterloo, ON, Canada
[2] Inria, Paris, France
[3] Cryptology Group, CWI, Amsterdam, The Netherlands

Abstract. Recent works have shown that quantum period-finding can be used to break many popular constructions (some block ciphers such as Even-Mansour, multiple MACs and AEs...) in the superposition query model. So far, all the constructions broken exhibited a strong algebraic structure, which enables to craft a periodic function of a single input block. Recovering the secret period allows to recover a key, distinguish, break the confidentiality or authenticity of these modes.
In this paper, we introduce the *quantum linearization attack*, a new way of using Simon's algorithm to target MACs in the superposition query model. Specifically, we use inputs of multiple blocks as an interface to a function hiding a linear structure. Recovering this structure allows to perform forgeries.
We also present some variants of this attack that use other quantum algorithms, which are much less common in quantum symmetric cryptanalysis: Deutsch's, Bernstein-Vazirani's, and Shor's. To the best of our knowledge, this is the first time these algorithms have been used in quantum forgery or key-recovery attacks.
Our attack breaks many parallelizable MACs such as LightMac, PMAC, and numerous variants with (classical) beyond-birthday-bound security (LightMAC+, PMAC+) or using tweakable block ciphers (ZMAC). More generally, it shows that constructing parallelizable quantum-secure PRFs might be a challenging task.

Keywords: Quantum cryptanalysis, MACs, superposition query model, Deutsch's algorithm, Bernstein-Vazirani algorithm, Simon's algorithm, Shor's algorithm.

1 Introduction

The possible emergence of large-scale quantum computing devices in a near future has prompted a wide move towards *post-quantum security*, which takes into account the new security threats that they pose. In particular, the most popular asymmetric cryptosystems currently in use, such as RSA, can be broken

by an adversary capable of successfully implementing Shor's algorithm [59]. An ongoing standardization project led by the NIST [56] has structured the efforts of the (asymmetric) cryptographic community on this question.

As symmetric primitives do not rely on a trapdoor, they seemed for a long time to avoid the cases where quantum computers bring an exponential speedup over the best classical algorithms. In fact, most problems in symmetric cryptography, such as the search for the secret key of a black-box cipher, seem to admit a quadratic speedup at best, given by Grover's quantum search algorithm [29]. Although this speedup is significant, it could be countered by increasing the parameters of symmetric cryptosystems, e.g., doubling the size of secret keys.

However, in the past few years, a series of works have shown the insecurity of some symmetric cryptosystems against quantum adversaries entitled to *superposition queries*. That is, some primitives become broken if they can be queried inside a quantum algorithm. This started with the 3-round Feistel distinguisher proposed by Kuwakado and Morii [44]. Later, they found a polynomial-time key-recovery attack on the Even-Mansour cipher [45], which was the first quantum key-recovery on a classically secure symmetric construction. These results rely crucially on the fact that many popular designs in symmetric cryptography have a strong algebraic structure, as they are built by combining smaller primitives (such as permutations or block ciphers) using cheap operations such as XORs. Kaplan *et al.* [39] showed that many other constructions exhibited a structure exploitable by a quantum adversary, and designed the first forgery attacks on MACs (notably CBC-MAC [11], OMAC [35], PMAC [12]) and authenticated encryption schemes (e.g., OCB3 [43], GCM [49]).

In this paper, we will focus on idealized MAC constructions that authenticate messages of arbitrary size using smaller primitives such as permutations, block ciphers or tweakable block ciphers (TBCs) of block size n. These constructions have classical proofs of security showing either that the MAC behaves as a pseudo-random function, or that it is unforgeable, up to some exponential bound in n. We will exhibit polynomial-time quantum attacks on constructions that were not vulnerable to previous Simon's attacks (like those of [39,58]).

Previous Attacks. Although there have been many of them, all the quantum forgery attacks known so far follow the same paradigm. They query the MAC with a constant number of blocks, using usually a single block of message x in superposition. Inside the MAC, this block of message x is XORed to some unknown value α depending on other blocks: thus, the result is $\mathsf{MAC}(x \oplus \alpha)$. Having two different values α_0, α_1, we then have access to two functions $f(x) = \mathsf{MAC}(x \oplus \alpha_0)$ and $g(x) = \mathsf{MAC}(x \oplus \alpha_1)$, such that $f(x) = g(x \oplus \alpha_0 \oplus \alpha_1)$. From there, we can use Simon's Boolean hidden shift algorithm [60] as a black box. It recovers $\alpha_0 \oplus \alpha_1$ in quantum polynomial time, whereas any classical algorithm would require exponentially many queries to f and g (thus to the MAC). The recovery of the internal shift $\alpha_0 \oplus \alpha_1$ then enables the adversary to forge new messages, and in some cases to recover secret-key material.

Let us point out the following important remark:

If the message blocks are not directly XORed to internal values (keys, offsets, encryption of other blocks...), then the previous attacks based on Simon's algorithm do not apply.

Contributions. In this paper, we present the *quantum linearization attack*, which is a new family of quantum attacks on classically unforgeable MACs when superposition queries are allowed. Thanks to the novelty of our approach, we are able to attack many MACs that resisted previous cryptanalysis, as they do not not exhibit the property recalled above (a message block XORed to an internal state value). In particular, our attack usually circumvents the use of TBCs instead of block ciphers. It is also the first case of a quantum polynomial-time attack on MACs with *beyond-birthday* security, where the internal state has a bigger size. As an example, we break LightMAC with a linear number of queries, and we can attack LightMAC+ with only twice as much.

Overview. Our attack starts with the following remark. Consider a function of ℓ blocks $x_1, \ldots, x_\ell$ of the form: $G(x_1, \ldots, x_\ell) = g_1(x_1) \oplus \ldots \oplus g_\ell(x_\ell) \oplus C$, where C is an independent constant, and the g_i are independent random functions to which the adversary *does not have access*. Then classically, this function cannot be distinguished from random with a single query, though as little as four would be enough: we make $x_3, \ldots, x_\ell$ constant, we query for every $x_1 \in \{0,1\}$ and $x_2 \in \{0,1\}$: the XOR of the four results is zero.

Our key idea is to *linearize* the function G by restricting the block inputs so that the output is an affine function. Similarly to the simple classical distinguisher, we make the blocks $x_1, \ldots, x_\ell$ take only one-bit values and emulate a function of an ℓ-bit input: $F(x) = F(b_1 \| \ldots \| b_\ell) = G(0^{n-1} \| b_1, \ldots, 0^{n-1} \| b_\ell)$. Now, we will remark that F is an *affine* function of $b_1, \ldots, b_\ell$. As the g_i are XORed; flipping a bit b_i in the input XORs $g_i(0) \oplus g_i(1)$ to the output.

It is well known that the Bernstein-Vazirani algorithm allows to distinguish an affine function from a random one with a *single* quantum query. This shows that, thanks to a multi-block input, we can access new vulnerabilities of cryptographic constructions. But the power of our attack is clearly demonstrated when we make G go through a new random function:

$$G'(x_1, \ldots, x_\ell) = g(G(x)) = g\left(g_1(x_1) \oplus \ldots \oplus g_\ell(x_\ell) \oplus C\right) \quad . \tag{1}$$

All the functions $g_1, \ldots, g_\ell, g$ are unknown to the adversary, so she cannot find the affine structure of the internal G. In fact, this function would be classically secure as a MAC. However, when linearizing, we obtain: $G'(x) = g(F(x))$ where F is an affine function of $x = b_1 \| \ldots \| b_\ell$. Thus, G' embeds a *hidden Boolean period*, and Simon's algorithm can recover it in polynomial time.

Applications. In Section 4 and Section 5, we detail the applications of our algorithm. We obtain the first polynomial-time attacks against the following MACs:

ΘCB3 [57,43], LightMAC [47], LightMAC+ [54], Deoxys [38],
ZMAC [37], PMAC_TBC3k [53], PolyMAC [36], GCM-SIV2 [36]

In addition, we provide attacks on the XOR-MACs of [4], on MACs based on universal hashing (e.g., NMH* [31] and BRW Hashing [7]) and, in Section 5.5,

a new superposition forgery attack against Poly1305 [6]. A previous quantum attack was given in [18], using a hidden shift structure. Using Shor's algorithm instead, we reduce the number of superposition queries from 2^{38} to about 32.

On Parallelizable MACs. The quantum linearization attack leaves only little space for quantum-secure parallelizable PRFs. Indeed, we are able to break any PRF with extendable domain, where at least $\geq n$ independent input blocks of $\leq n$ bits are processed independently, then XORed. This works as well for any operation that is linear on $(\mathbb{F}_2)^n$. It is still possible to obtain an unforgeable IV- or nonce-based MAC of this form, as shown in [9], but the security then relies on the non-repetition of IVs. We do not know if an attack applies when we use a modular addition instead of a XOR in (1). If this was the case, then it would clearly mean that one has to rely on sequentiality or on nonlinear operations.

Organization. We start in Section 2 by reviewing some quantum computing notions, the quantum algorithms used in this paper (Deutsch's algorithm, Bernstein-Vazirani, Simon's algorithm, Shor's algorithm), the Q1 / Q2 attack scenarios and notions of quantum unforgeability. In Section 3, we detail our new algorithmic ideas. In Section 4, we apply our attack to many parallelizable MAC constructions. We dedicate Section 5 to MACs based on universal hashing. We discuss the implications of our attacks in Section 6 and conclude the paper in Section 7.

2 Preliminaries

In this section, we give some preliminaries about quantum computing, quantum attacker models and the well-known quantum algorithms that will be used throughout this paper. We elaborate about the Q2 attacker model and the notion of quantum unforgeability for MACs, with or without IVs. Note that some details of quantum computing will appear in this section. They are intended for the interested reader. In the rest of this paper (with the exception of Section 5.5), we will use the algorithms of this section as black boxes.

2.1 Notation

We consider n-bit string values, sometimes as elements of $\mathbb{F}_{2^n}$, sometimes as elements of $\mathbb{F}_2^n$. This shall be clear from context. We let $\oplus$ denote the XOR (addition in $\mathbb{F}_2^n$), $\odot$ denote multiplication in $\mathbb{F}_{2^n}$, and $+$ modular addition. We let $\cdot$ denote the scalar product of bit-strings seen as n-bit vectors.

2.2 On Quantum Computing

Although we choose to present in detail the quantum algorithms that we will use for our attacks, most of our results can be obtained by applying them as black boxes. Thus we stress that our results, similarly as other structural attacks on symmetric cryptosystems [45,39], can be understood from a high-level perspective, and our attacks do not require specific knowledge of quantum computing. Further details are only required to prove the correctness of the algorithms.

A general presentation of the *quantum circuit model* can be found in [55]. The basic computation units are qubits, two-level quantum systems whose state is represented by a *superposition* $\alpha|0\rangle + \beta|1\rangle$, with amplitudes α and β, which is a normalized vector in $\mathbb{C}^2$ (of norm $|\alpha^2| + |\beta^2| = 1$). The state of an n-qubit system belongs to $\mathbb{C}^{2^n}$, its 2^n basis vectors (in the computational basis) are the 2^n n-bit strings.

A quantum algorithm is a sequence of unitary operators of $\mathbb{C}^{2^n}$, partial measurements, and oracle calls. We say that a function f is queried *in superposition* if the following unitary operator O_f is made available: $|x\rangle|y\rangle \mapsto |x\rangle|y \oplus f(x)\rangle$. Indeed, this operator allows to query f on any quantum state, thus on any *superposition* of inputs x. This is the *standard* oracle, equivalent to the *phase oracle* $O_{f,\pm}$ which computes $|x\rangle \mapsto (-1)^{f(x)}|x\rangle$.

One of the basic unitary operations of the quantum circuit model (*quantum gates*), and actually the most important one in the algorithms of Section 2.3, is the Hadamard gate H which maps a single qubit $|b\rangle$ to $\frac{1}{\sqrt{2}}\left(|0\rangle + (-1)^b|1\rangle\right)$. By applying Hadamard gates to each individual qubit of an n-bit input, we compute the *Hadamard transform*, a particular example of Quantum Fourier Transform:

$$H^{\otimes n} : |x\rangle \mapsto \frac{1}{2^{n/2}} \sum_{y\in\{0,1\}^n} (-1)^{x\cdot y}|y\rangle \quad .$$

An important property is that the Hadamard transform is involutive. For better readability, we often omit global amplitude factors such as the $\frac{1}{2^{n/2}}$ above, as quantum states are always normalized.

Given a quantum state of the form $\sum_x \alpha_x|x\rangle$, the measurement operation destroys the state and yields an element x with probability $|\alpha_x|^2$. Partially measuring the state *projects* it on a smaller superposition of elements. For a quantum state of the form: $\sum_{x,y} \alpha_{xy}|x\rangle|y\rangle$, measuring the register $|x\rangle$ yields a value x_0 with probability $\sum_y |\alpha_{x_0 y}|^2$, and projects on the state $\frac{1}{\sqrt{\sum_y |\alpha_{x_0 y}|^2}} \sum_y \alpha_{x_0 y}|y\rangle$.

2.3 Quantum Algorithms

Our new attacks are based on well-known quantum algorithms: Deutsch's algorithm [26], which is a single-bit version of the Deutsch-Jozsa algorithm [27], the Bernstein-Vazirani algorithm [8], Simon's algorithm [60] and Shor's algorithm [59]. These algorithms have in common to be based on *Fourier sampling*, a process in which a quantum Fourier transform is applied before and after a single query to a superposition oracle. They are also amongst the earliest quantum algorithms proven to beat any classical algorithm, and as such are often presented in textbooks (see e.g. [55]). However, except for Shor's algorithm, their practical interest remained unclear for a long time.

Deutsch's Algorithm. Deutsch's algorithm [26] solves Problem 1 with probability 1 using a single query to O_f, whereas classically, two queries to f are needed for the same success probability. This constant speedup might seem anecdotal, but is crucial when the same function cannot be queried more than once.

Algorithm 1 Deutsch's algorithm

1: Start from $|0\rangle$ ▷ $|0\rangle$
2: Apply a Hadamard gate ▷ $|0\rangle + |1\rangle$
3: Apply $O_{f,\pm}$ ▷ $(-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle = (-1)^{f(0)}\left(|0\rangle + (-1)^{f(0)\oplus f(1)}|1\rangle\right)$
4: Apply a Hadamard gate ▷ $(-1)^{f(0)}|f(0)\oplus f(1)\rangle$
5: Measure the state

Algorithm 2 Bernstein-Vazirani algorithm

1: Start from $|0_n\rangle$ ▷ $|0_n\rangle$
2: Apply a Hadamard transform ▷ $\sum_i |i\rangle$
3: Apply $O_{f,\pm}$ ▷ $\sum_i (-1)^{(a\cdot i)\oplus b}|i\rangle$
4: Apply a Hadamard transform
5: Measure the state ▷ $(-1)^b H^{\otimes n}\sum_i (-1)^{a\cdot i}|i\rangle = (-1)^b H^{\otimes n}\left(H^{\otimes n}|a\rangle\right) = (-1)^b|a\rangle$

Problem 1 (Deutsch's problem). Given access to a quantum oracle O_f for a function $f : \{0,1\} \to \{0,1\}$, decide whether f is *constant* ($f(0) = f(1)$) or *balanced* ($f(0) \neq f(1)$).

Deutsch's algorithm (Algorithm 1) is best presented with a phase oracle $O_{f,\pm}|b\rangle = (-1)^{f(b)}|b\rangle$. It can be seen that upon measurement, the algorithm actually yields the value $f(0) \oplus f(1)$ (although a single query has been made to f) whose knowledge solves Problem 1.

Bernstein-Vazirani Algorithm. The Bernstein-Vazirani algorithm [8] offers a polynomial speedup for finding the slope of an affine function over $\mathbb{F}_2^n$.

Problem 2 (Bernstein-Vazirani). Given access to an oracle O_f for an affine function $f : \{0,1\}^n \to \{0,1\}$, that is, $f(x) = a \cdot x \oplus b$ for a, b unknown, find a.

Upon measurement in Algorithm 2, we obtain the unknown a with certainty, using a single query to $O_{f,\pm}$, while n queries would be needed classically.

Remark 1. This algorithm can be seen as a generalization of Deutsch's algorithm. Indeed, in the case $n = 1$, there are only two types of affine functions: $f(x) = x \oplus b$ ($a = 1$) and $f(x) = b$ ($a = 0$), and Bernstein-Vazirani allows to distinguish them in one query.

Simon's Algorithm. Simon's algorithm [60] solves the problem of distinguishing an injective function from a periodic one. Note that it was the first example of an exponential quantum speedup relatively to an oracle.

Problem 3 (Simon). Given access to a function $f : \{0,1\}^n \to \{0,1\}^n$ for which there exists s such that: $\forall x, y, f(x) = f(y) \iff y \in \{x, x \oplus s\}$, find s.

In Algorithm 3, at Step 9 in the injective case, the value a obtained before has a single preimage x_a. Thus, the current state is $\sum_y ((-1)^{x_a \cdot y})|y\rangle$ and we sample a uniformly random $y \in \{0,1\}^n$. After $n + r$ such samples, the family Y

Algorithm 3 Simon's algorithm

$Y = \emptyset$
Choose a number r depending on the required probability of error
Repeat $n + r$ **times**
    Start from $|0_n 0_n\rangle$
    Apply a Hadamard transform to the first register ▷ $\sum_x |x\rangle |0\rangle$
    Apply O_f (standard) ▷ $\sum_x |x\rangle |f(x)\rangle$
    Measure the second register, obtain a ▷ $\sum_{x|f(x)=a} |x\rangle$
    Apply a Hadamard transform ▷ $\sum_y (\sum_{x|f(x)=a} (-1)^{x\cdot y}) |y\rangle$
    Measure a y, $Y \leftarrow Y \cup \{y\}$
EndRepeat
if Y is of full rank **then**
    return "injective case"
else if Y is of rank $n-1$ **then**
    return "periodic case" and the s orthogonal to Y
else
    return "failure"
end if

will grow to a full-rank family. In the periodic case, the value a has exactly two preimages x_a and $x_a \oplus s$ which interfere with each other. The current state is

$$\sum_y ((-1)^{x_a \cdot y} + (-1)^{(x_a \oplus s)\cdot y}) |y\rangle = \sum_y (-1)^{x_a \cdot y} (1 + (-1)^{s \cdot y}) |y\rangle$$

and only the vectors y orthogonal to s have a non-zero amplitude. Thus, the family Y grows to span the euclidean subspace orthogonal to s. Computing the rank of Y allows to detect the period and solving the linear system $Ys = 0_n$ allows to recover it.

Generalizations. Although the original Simon's problem concerns functions without random collisions (that is, we cannot have $f(x) = f(y)$ if $x \oplus y \notin \{0, s\}$), it can be shown that the algorithm works as well for *random functions having a period*, which models the cryptographic problems that we are interested in.

The following simple condition was given in [39]. For Simon's algorithm to run as expected (i.e., with $\mathcal{O}(n)$ queries), it is sufficient for the periodic function f, of period s, to satisfy the following condition:

$$\max_{t \notin \{0,s\}} \Pr_x [f(x \oplus t) = f(x)] \leq \frac{1}{2} \ . \tag{2}$$

That is, f should not admit another "unwanted partial period" t. In the examples studied in this paper, the condition (2) will be easy to check.

Note that if we had $f(x \oplus t) = f(x)$ for all x, then t would simply turn the set of periods of f into a vector space of dimension 2. In general, the space of periods could be a vector space of any dimension. An extended version of Simon's algorithm by Brassard and Høyer [20] allows to recover this whole space in polynomial time.

Finally, another important case is when the output set is smaller than the input set. This was studied in [16] for Simon's algorithm and [48] for period-finding in general. The results in [16] show that as long as the functions behave as random (but with the periodicity constraint), then for n input bits, the number of output bits required to run correctly without any cost increase is of order $\log_2 n$. The results in [48] show that the output can be hashed down to a single bit, and the algorithms still work up to a constant increase in queries.

Shor's Algorithm. We will use Shor's algorithm [59] to solve the *abelian hidden period* problem. It will appear in a "black-box" manner in Section 5.4, and in Section 5.5. We will analyse in detail the behavior of the algorithm on Poly1305.

Problem 4 (Abelian hidden period). Let $(G, +)$ be an abelian group, X a set. Given access to a function $f : G \to X$ which is either injective, or periodic ($\exists s \in G, f(s + \cdot) = f(\cdot)$), then determine the case and / or find the period.

In particular, we consider $G = \mathbb{Z}_{M_1} \times \ldots \times \mathbb{Z}_{M_k}$ the product of multiple cyclic groups of known order. For simplicity, and to prepare for Section 5.5, we present the algorithm in the case of $\mathbb{Z}_p^2$ for some prime p. Note that f is also periodic over $\mathbb{Z}_p$ in each of its parameters. This is the typical situation when Shor's algorithm is used to solve the Discrete Logarithm Problem. The periods of f form a two-dimensional integer lattice, which is generated by $(p, 0)$ and $(-1, s)$ for some s. In other words, the value of $f(x, y)$ depends only on the value of $xs + y \bmod p$. We may assume for simplicity that the function $xs + y \bmod p \mapsto f(x, y)$ is injective.

The algorithm only relies on an efficient implementation of the Quantum Fourier Transform over $\mathbb{Z}_p$:

$$|x\rangle \mapsto \sum_{y=0}^{p-1} \exp\left(2i\pi \frac{xy}{p}\right) |y\rangle \quad ,$$

which we assume exact. We represent the elements of X on m bits.

In Algorithm 4, at Step 4, we can only measure a vector $|z, t\rangle$ having a nonzero amplitude. This means that we need:

$$\sum_{x=0}^{p-1} \exp\left(2i\pi \frac{(z - st)x}{p}\right) \neq 0 \quad ,$$

which happens only when $(z - st) = 0$. In that case, the sum simply gives p. After renormalization, all vectors $|z, t\rangle$ with $(z - st) = 0$ have the same amplitude $\frac{1}{\sqrt{p}}$, and we will measure one of them taken uniformly at random. If $t \neq 0$, we compute s by $s = zt^{-1} \bmod p$. This occurs with probability $1 - \frac{1}{p}$.

2.4 Attack Scenarios

We consider different *attack scenarios* throughout this paper.

Algorithm 4 Shor's algorithm

1: Start from $|0,0,0_m\rangle$ ▷ $|0,0,0_m\rangle$
2: Apply a Quantum Fourier Transform on both input registers ▷ $\sum_{x,y=0}^{p-1} |x,y\rangle\,|0_m\rangle$
3: Apply O_f ▷ $\sum_{x,y=0}^{p-1} |x,y\rangle\,|f(x,y)\rangle$
4: Measure the second register. The state collapses on a uniform superposition of all (x,y) such that $xs+y=a \bmod p$ for some unknown a, meaning $y=a-xs \bmod p$:

$$\sum_{x=0}^{p-1} |x\rangle\,|a-xs\rangle \quad .$$

5: Apply a Quantum Fourier Transform again. The state becomes:

$$\sum_{z,t=0}^{p-1}\left(\sum_{x=0}^{p-1}\exp\left(2i\pi\frac{zx+(a-xs)t}{p}\right)\right)|z,t\rangle$$
$$=\sum_{z,t=0}^{p-1}\exp(2i\pi at/p)\left(\sum_{x=0}^{p-1}\exp\left(2i\pi\frac{(z-st)x}{p}\right)\right)|z,t\rangle \quad .$$

6: Measure a $|z,t\rangle$ and return $s=zt^{-1} \bmod p$.

Q1 and Q2 setting. Following [40,33,17], we will adopt the Q1 / Q2 terminology to classify quantum attacks on symmetric schemes. Note that these models have alternative names, for example "quantum chosen-plaintext attack" (qCPA) is used for "Q2" in [34,22]. In the Q1 setting, the adversary is given only *classical* encryption or decryption query access to black-boxes. In the Q2 setting, the adversary is given *quantum* or *superposition* access, in the sense that a black-box E_K becomes a quantum oracle O_{E_K}. This is the case for all the attacks of this paper.

The study of quantum attacks on symmetric schemes in the Q2 setting was sparkled by seminal work of Kuwakado and Morii [44,45], who showed that the 3-round Luby-Rackoff construction and the Even-Mansour cipher became insecure if exposed to superposition queries. More precisely, they can use Simon's algorithm to respectively distinguish the construction and recover the key of the cipher in polynomial time, while classical proofs of security exist.

Attacks based on period-finding. Since these earlier results, many works have extended the reach of Q2 attacks [39,15,46,19,28,30]. However, the attack strategy has remained the same. A hidden structure is embedded in the construction to be attacked, so that $f(E_K(x),x)$ for some choice of combination f, is a periodic function of x; or that a shift exists between $f(E_K(x),x)$ and $g(E_K(x),x)$. The recovery of this hidden period or shift, which is secret material, then leads to a break. We can cite some examples:

Against the Even-Mansour construction [45]: $E_K(x) = K_2 \oplus \Pi(x \oplus K_1)$ for a random public permutation Π and two keys K_1, K_2. One has:

$$E_K(x) \oplus \Pi(x) = E_K(x \oplus K_1) \oplus \Pi(x \oplus K_1)$$

which leads to a recovery of K_1 in $\mathcal{O}(n)$ queries and $\mathcal{O}\left(n^3\right)$ computations.

Against CBC-MAC *with two blocks [39]:* It can be defined as:

$$\mathsf{CBC\text{-}MAC}(y, x) = E_{K'} \circ E_K\Big(x \oplus E_K(y)\Big) \ ,$$

where K and K' are two keys that will remain unknown to the adversary. Due to the structure of CBC-MAC, one can take two arbitrary values α_0, α_1, and define the function:

$$F \ : \begin{cases} \{0,1\} \times \{0,1\}^n \mapsto & \{0,1\}^n \\ (b, x) & \to \mathsf{CBC\text{-}MAC}(\alpha_b, x) \end{cases} \tag{3}$$

We have then that $F(b, x) = F(b \oplus 1, x \oplus E_K(\alpha_0) \oplus E_K(\alpha_1))$. Thus F has a hidden boolean period $1 \| E_K(\alpha_0) \oplus E_K(\alpha_1)$. Having obtained the internal value $E_K(\alpha_0) \oplus E_K(\alpha_1)$, we can query the tag of any message starting with block α_0, and then forge a message starting with α_1 with the same tag.

Constructions based on IVs. We consider two types of constructions with quantum access: some make use of an *initialization value* (IV, sometimes also named a *nonce*) and some do not. In the IV case, we consider that the IV is a classical value, chosen randomly before each oracle query, and not repeated. This model follows from the idea that the IV is not controlled by the adversary, and it can serve as an intermediate between the classical setting and a (much) stronger model in which the adversary would completely (and quantumly) control the IVs.

In fact, the latter case does not seem to have been studied so far in quantum security. Well-known notions such as IND-qCPA [14] rely on classical randomness, and many modes of operation have been proven secure in this model [3,9].

In the classical setting, many MAC constructions have a security that relies on the non-repetition of IVs, for example the MAC of OCB [43]. The same happens in the quantum setting, since the MAC of QCB [9] has been proven secure under quantum queries with classical non-repeated IVs.

Unforgeability. The first notion of quantum unforgeability for MACs was defined by Boneh and Zhandry [13]. We will name it *plus-one unforgeability* (PO), following [1]. The idea is that an adversary making q quantum queries to the construction, where q is polynomial, should not be able to produce $q + 1$ valid $\{\text{message}, \text{tag}\}$ pairs. A more recent definition is *blind-unforgeability* (BU), proposed in [1]. It is strictly stronger than PO-unforgeability. In this paper, we will give several quantum forgery attacks that break the PO notion, thereby also breaking BU.

Quantum PRFs. A *quantum pseudorandom function* (qPRF) is a family of functions F_K, indexed by a key space $\mathcal{K}$, such that no quantum adversary making queries to an oracle O_f can distinguish efficiently between a function F_K, with

K drawn uniformly at random, and a truly random function. It is shown respectively in [13] and [1] that a qPRF is also a quantum-secure deterministic MAC by the PO and BU definitions. Therefore, any function that is not PO-unforgeable is also not a secure qPRF. To the best of our knowledge, the only classical symmetric construction that has been proven quantum-secure as a deterministic MAC, the Cascade / NMAC / HMAC construction [61], is also a qPRF.

2.5 A Quantum Attack on OCB3

We detail the Q2 attack on the MAC of OCB3 from [39]. As the other previous works recalled above, this attack relies on a Boolean period-finding problem.

Specification. OCB3 is an IV-based mode of authenticated encryption with associated data (AEAD), based on a block cipher [43]. As OCB stands for *offset codebook*, the scheme relies on the definition of *offsets* that are dependent on the key and change between each block. We will focus on the authentication tag of OCB3 (see Figure 1). Our considerations are independent on the exact value of the offsets, and apply to all versions of OCB, but we use OCB3 as a concrete example.

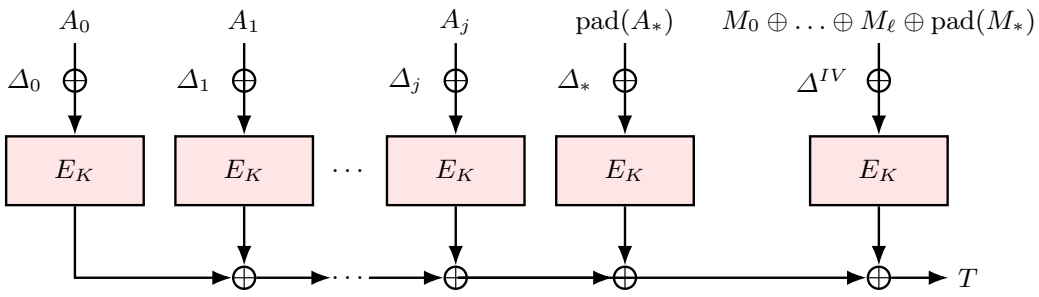


Fig. 1. Computation of the tag in OCB3. Only the offset Δ^{IV} depends on the IV.

Forgery Attack with Simon's Algorithm. Kaplan *et al.* showed in [39] how to forge authentication tags using Simon's algorithm. The idea is to query the tag of an empty message with two AD blocks $A_0, A_1 = x$:

$$x \to E_K(\Delta_{IV}) \oplus E_K(\Delta_0 \oplus x) \oplus E_K(\Delta_1 \oplus x) \ .$$

One can then remark that this function of x is periodic, of period $\Delta_0 \oplus \Delta_1$, independent of the IV, and only on the secret key K. Although the function changes at each query (since the IV changes), the period is always the same and Simon's algorithm allows to recover it with $\mathcal{O}(n)$ superposition queries. (For the same reason, we could use a non-empty message, and even different messages between the queries.)

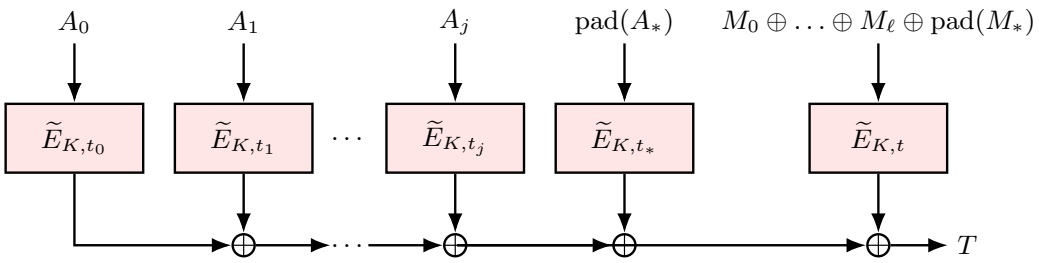


Fig. 2. Computation of the tag in ΘCB3. Only the final tweak t depends on the IV.

Once $\Delta_0 \oplus \Delta_1$ has been obtained, one can then query the tag of any pair of AD blocks A_0, A_1 and forge the tag of A_1, A_0.

Remark 2. It is easy to check that Equation (2) is satisfied in practice. If it wasn't, then the existence of an unwanted partial period t:

$$\Pr_x \left[f(x \oplus t) = f(x) \right] \geq \frac{1}{2} \ ,$$

would imply a higher-order differential of probability greater than $\frac{1}{2}$ for E_K, which is impossible if E_K is a pseudorandom permutation (in other words, E_K would suffer from a classical break).

3 The Quantum Linearization Attack: Algorithmic Ideas

In this section, we present the algorithmic ideas underlying our new *quantum linearization attack*. To that end, we keep the example of OCB3 [43] introduced in Section 2.5. We explain a new way to forge with Q2 queries. The extensions and applications of this new idea will be explored in the next sections.

Note that to the best of our knowledge, this is the first application of the Deutsch and Bernstein-Vazirani algorithms for forgery attacks.

3.1 Attack on ΘCB with Deutsch's Algorithm

The attack of Section 2.5 works only because of the offsets. In fact, the existence of a controlled value (here x) XORed to a secret (here the offsets) has been so far a prerequisite of all Q2 attacks.

Here we present a forgery attack against the mode ΘCB3 [57,43], which is a more abstract version of OCB3 in which the block cipher E_K is replaced by a *tweakable* block cipher (a family of independent block ciphers $\widetilde{E}_{K,t}$ indexed by a *tweak* t). This is shown in Figure 2.

Here, the tweaks $t_0, \ldots, t_j, t_*$ form an arbitrary sequence of distinct values, that depend only on the block index; the tweak t is the only one dependent on the IV. Again, we consider an empty message, but this time a single AD block

that is either 0 or 1. We define i functions which truncate the output of such a call to the ith bit:

$$F_i : \begin{array}{ccc} \{0,1\} & \to & \{0,1\} \\ b & \mapsto & \mathrm{Trunc}_i(\widetilde{E}_{K,t_0}(b) \oplus \widetilde{E}_{K,t}(0)) \end{array} .$$

The functions F_i change at each new superposition query (because the IV intervenes in $\widetilde{E}_{K,t}(0)$). Thus we need the ability to compute a query to F_i using a *single* query to the untruncated mode itself. This is fortunately easy to do so using the truncation technique of [33].

With this single query, Deutsch's algorithm allows to recover the value:

$$\mathrm{Trunc}_i(\widetilde{E}_{K,t_0}(0) \oplus \widetilde{E}_{K,t}(0)) \oplus \mathrm{Trunc}_i(\widetilde{E}_{K,t_0}(1) \oplus \widetilde{E}_{K,t}(0)) \\ = \mathrm{Trunc}_i(\widetilde{E}_{K,t_0}(0) \oplus \widetilde{E}_{K,t_0}(1)) \ ,$$

and within n queries and uses of the algorithm, we can obtain the full value $\widetilde{E}_{K,t_0}(0) \oplus \widetilde{E}_{K,t_0}(1)$.

We can now forge valid messages as follows: we query a message with 0 as the first block, we XOR $\widetilde{E}_{K,t_0}(0) \oplus \widetilde{E}_{K,t_0}(1)$ to the tag, and we have obtained the tag of the same message with 1 replacing the first block. This works for any block and for any pair of messages.

This attack shows that XORing with an IV-dependent value, although it provides sufficient protection against forgeries in the classical setting (since ΘCB has a security proof), does not in the quantum setting.

Interestingly, it is possible to protect against this attack by using the IV in the TBC calls, as done by Bhaumik et al. in [9]. While this simple modification has practically no incidence on the classical security of the mode, it is crucial to obtain unforgeability in the quantum setting.

Another Example: XOR-MACs. In [4], two *XOR-MAC* constructions are defined, which can be attacked with Deutsch's algorithm. They are both based on a pseudorandom function F_K and an IV. The first one, XMACR (*randomized XOR scheme*), considers that the IV is drawn uniformly at random, and the second one, XMACC (*counter-based XOR scheme*) that it is maintained as a counter. Both compute:

$$\mathsf{MAC}(m_1, \ldots, m_\ell; IV) = F_K(0\|IV) \oplus \bigoplus_{1 \le i \le \ell} F_K(1\|i\|m_i) \ .$$

Then, since the contribution of the IV is only XORed, forgeries can be made.

3.2 Using the Bernstein-Vazirani algorithm

We propose here a generalization of the previous attack, with longer queries. We now consider functions of the form

$$g_1(x_1) \oplus g_2(x_2) \oplus \cdots \oplus g_\ell(x_\ell) \oplus C$$

with, as before, a C that is independent from all x_i. Now, we can choose some arbitrary α_i^0 and α_i^1, and consider the function

$$F_j : \begin{array}{ccc} \{0,1\}^\ell & \to & \{0,1\} \\ (b_1,\ldots,b_\ell) & \mapsto & \mathrm{Trunc}_j\left(\bigoplus_{i=1}^{\ell} g_i(\alpha_i^{b_i}) \oplus C\right) \end{array} ,$$

It is easy to see that this function is affine: indeed, if we change the value of b_i, then we add $\mathrm{Trunc}_j\left(g_i(\alpha_i^0) \oplus g_i(\alpha_i^1)\right)$ to the output.

Hence, if we apply the Berstein-Vazirani algorithm, in one query, we recover the values of the $\mathrm{Trunc}_j\left(g_i(\alpha_i^0) \oplus g_i(\alpha_i^1)\right)$, for all i. Next, it suffices to repeat the algorithm for each bit of the output to obtain the value of all the $g_i(\alpha_i^0) \oplus g_i(\alpha_i^1)$.

This technique can be applied to OCB3 / ΘCB3, as the tag is a function of the form

$$\bigoplus_i g_i^k(AD_i) \oplus f_k(IV, M)$$

Hence, we can attack multiple blocks of associated data at once.

3.3 Attacking any XOR of permutations

The main limitation of the previous attacks is that they need a direct access to the linear combination of independent blocks. In this section, we overcome this limitation with an attack that leverages linear combinations of permutations in a more intrinsic way, using Simon's algorithm in a novel fashion.

We consider a MAC construction that processes $m > n$ message blocks $x_1, \ldots, x_m$ by pushing the x_i through independent TBC calls $\widetilde{E}_{K,i}$, XORing the result and applying an IV-dependent function afterwards.

$$IV, (x_1, \ldots, x_m) \mapsto f_K\left(IV, \left(\bigoplus_{1 \le i \le m} \widetilde{E}_{K,i}(x_i)\right)\right) .$$

Remark 3. We write the attack with a TBC, i.e., a family of independent block ciphers $\widetilde{E}_{K,T}$ indexed by a secret key K *and* a public tweak T. This is to emphasize the application of our attack to parallelizable MACs; however, the attack works in the same way if we replace the independent block ciphers by independent functions.

In the case of ΘCB, the definition of f_K is simple, since it only XORs the IV- and the AD-dependent parts. But the attack of Section 3.2 does not apply anymore if f_K is a pseudorandom function. This will be the case of our new attack, which is why it will apply to many MAC constructions.

Quantum Attack. First of all, it is easy to see that if the $\widetilde{E}_{K,i}$ are independent block ciphers, and if f_K is a pseudorandom function family, then this construction is a classically unforgeable MAC: this is the security of ΘCB3.

Our attack in the quantum setting starts from the same idea as above (Section 3.2): we query the MAC with arbitrary blocks taking two values: $x_1 =$

$b_1||0_{n-1}$, ..., $x_m = b_m||0_{n-1}$, where $x = b_1 \dots b_m$ forms an m-bit input (in the remaining of this paper, we will write the $n-1$ zeroes used for completion as a single 0). We will put x in superposition, and so, there will be only "one superposed bit" in each individual block input.

One then observes that $\widetilde{E}_{K,1}(x_1) \oplus \ldots \oplus \widetilde{E}_{K,m}(x_m)$ is an affine function of x:

$$F(x) := \widetilde{E}_{K,1}(x_1) \oplus \ldots \oplus \widetilde{E}_{K,m}(x_m) = \bigoplus_i \left(b_i \odot \left(\widetilde{E}_{K,i}(0) \oplus \widetilde{E}_{K,i}(1) \right) \oplus \widetilde{E}_{K,i}(0) \right) \ .$$

More precisely, if we identify bit-strings with boolean column vectors, $F(b_1 \dots b_m)$ is equal to:

$$\underbrace{\left((\widetilde{E}_{K,1}(0) \oplus \widetilde{E}_{K,1}(1)) \cdots (\widetilde{E}_{K,m}(0) \oplus \widetilde{E}_{K,m}(1)) \right)}_{M_m} \times \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} \oplus \bigoplus_i \widetilde{E}_{K,i}(0) \ .$$

The matrix M_m has n rows and m columns, so when $m \geq n+1$, its kernel is nontrivial. This means there will exist a non-zero m-bit boolean vector α such that:

$$\left((\widetilde{E}_{K,1}(0) \oplus \widetilde{E}_{K,1}(1)) \cdots (\widetilde{E}_{K,m}(0) \oplus \widetilde{E}_{K,m}(1)) \right) \times \alpha = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix} \ ,$$

and for all such vectors α, seen as m-bit strings, we have:

$$F(x \oplus \alpha) = M_m \times (x \oplus \alpha) \oplus \bigoplus_i \widetilde{E}_{K,i}(0) = F(x) \ .$$

In other words, this function F hides a subgroup of $(\mathbb{F}_2)^m$ generated by all the vectors α satisfying the condition above (it is easy to see that they indeed form a group). Thus, F satisfies the promise of Simon's algorithm: by making a single superposition query, we can find y such that $y \cdot \alpha = 0$ for such an α, and furthermore, as Brassard and Høyer showed [20], we can even recover the full subspace of periods with a polynomial number of quantum queries to F.

However, in our model, we cannot query F directly and we have instead access to: $f_K(IV, F(x))$, where IV changes at each query. The key remark is that the hidden subgroup is unchanged, since F is independent of the IV. This assumption is enough to allow Simon's algorithm and its extensions to work.

Remark 4 (Smaller m). Some period might still arise if $m \leq n$. Indeed, if $m = n$, there will be a non-trivial period with probability around $1 - 1/e$. This quickly decays for smaller m.

Remark 5 (Unwanted collisions). Since the "inner" function F is affine, it does not contain any unwanted collisions. If $F(x \oplus \alpha) = F(x)$ for some α and x,

then this holds as well for all x. However, unwanted collisions might occur in $f_K(IV, \cdot)$.

Assuming that M_m is full rank, we can express the probability of unwanted partial periods for $f_K(IV, F(\cdot))$ as the probability of such unwanted collisions for $f_K(IV, \cdot)$:

$$\begin{aligned} p &= \max_{t, M_m \times t \neq 0} \Pr_{x \in \{0,1\}^\ell} \left[f_K(IV, F(x \oplus t)) = f_K(IV, F(x)) \right] \\ &= \max_{t, M_m \times t \neq 0} \Pr_{x \in \{0,1\}^\ell} \left[f_K(IV, F(x) \oplus M_m \times t) = f_K(IV, F(x)) \right] \\ &= \max_{u \neq 0} \Pr_{x \in \{0,1\}^\ell} \left[f_K(IV, F(x) \oplus u) = f_K(IV, F(x)) \right] \\ &= \max_{u \neq 0} \Pr_{y \in \{0,1\}^n} \left[f_K(IV, y \oplus u) = f_K(IV, y) \right] \ . \end{aligned}$$

Even if the output is truncated to less than n bits, $p \leq \frac{1}{2}$ follows trivially from the fact that $f_K(IV, \cdot)$ should not admit a differential of such high probability. To conclude, it is precisely the fact that the termination function $f_K(IV, \cdot)$ is a good PRF, and does *not* admit an interfering period, that allows to apply easily Simon's algorithm in our case.

Thus, by making a polynomial number of Q2 queries to the MAC construction, we can obtain such an α. This allows to create forgeries as follows.

Forgeries without IVs. We first make n queries to find a valid α with Simon's algorithm (with constant probability of success). Then, the knowledge of this α allows us, for each tag x queried, to output a forged tag $x \oplus \alpha$. Thus we can double the number of tags that we produce compared to the number of queries we make. This breaks the PO notion as soon as, making $r + n$ queries, we have $2r \geq r + 1 + n$ tags, thus with $2n + 2$ queries in total. Note that by breaking PO, we are actually showing that the MAC construction is not a qPRF (if it were, it would be PO-secure).

Forgeries with IVs. As long as the IV (or nonce) is used only in the keyed post-processing, we can recover a value α and run the attack as above. We will indeed output more triples $\{IV, \text{message}, \text{tag}\}$ than the number of queries made, although some IVs are repeated in the outputs.

Universal Forgeries. Instead of taking the arbitrary values $b_i||0$ in message blocks, we can take any pair of values for each of them. That way, we can even start from any m-block message $y_1, \ldots, y_m$, and then define a function of $x = b_1 \ldots b_m$ that inputs y_i in block i if $b_i = 1$ and an arbitrary value 0 otherwise. Using Simon's algorithm, we will find a subset of the y_i such that the $\widetilde{E}_{K,i}(y_i)$ have the same XOR as the $\widetilde{E}_{K,i}(0)$. Hence, we can produce a new message having the same tag as $y_1, \ldots, y_m$. This works as soon as $m \geq n$ (there just needs to be enough message blocks for our attack).

4 Applications to Parallelizable MACs

In this section, we apply the quantum linearization attack to many parallelizable MACs of the literature. In particular, we show that the attack can be extended to

parallelizable beyond birthday-bound (BBB) MACs, although they have a larger internal state. Here is a summary of MACs attacked in this section (usually in time quadratic in the internal state size n), whose previous best quantum attack was exponential:

LightMAC [47], LightMAC+ [54], Deoxys [38], ZMAC [37], PMAC_TBC3k [53]

On the contrary, here are some MACs on which, to the best of our knowledge, our attack does not apply: SUM-ECBC [62], 2K-ECBC-Plus [24], 3kf9 [63]. The best Q2 attacks on these remain exponential-time (usually $\widetilde{\mathcal{O}}\left(2^{n/2}\right)$ or $\mathcal{O}\left(2^{k/2}\right)$ where n is the internal block size, and k the key size).

4.1 First Examples

We will consider MAC designs with or without IVs or nonces. When there is no IV, then the attack of Section 3.3 breaks them in the PO notion. This also shows that even though they usually yield classical PRFs, these constructions are not quantum-secure PRFs. When there is an IV, the MAC may be insecure as a PRF but still secure as a MAC (since the IV is changed at each query, and not repeated). Despite that, our attack may still yield a break, as we showed in the example of ΘCB above. In that case, the period that is recovered with Simon's algorithm is independent of the IV, and can be reused to forge a new valid (message, tag) pair under any previously queried IV.

LightMAC. LightMAC [47] is based on an n-bit block cipher and separates the message in blocks of $n-s$ bits, where $s \leq n/2$ is some parameter that limits the maximal message size. The function is the following:

$$\mathsf{LightMAC}(m_1, \ldots, m_\ell) = \mathrm{Trunc}_t\left(E_{K_2}\left((m_\ell 10*) \oplus \bigoplus_{i=1}^{\ell-1} E_{K_1}(i_s m_i)\right)\right) \ ,$$

where the i_s are s-bit constants. Calling LightMAC with single-bit blocks and using Simon's algorithm, we immediately obtain a sequence of indices $j_1, \ldots, j_v$ such that $E_{K_1}(i_{j_1}1) \oplus \ldots \oplus E_{K_1}(i_{j_v}1) = E_{K_1}(i_{j_1}0) \oplus \ldots \oplus E_{K_1}(i_{j_v}0)$ and thus, we can produce existential forgeries, and universal forgeries of messages with a linear number of blocks.

Deoxys. Due to the similarity of its MAC with ΘCB, our attack applies to all versions of Deoxys-II [38], one of the finalists of the CAESAR competition (it also applies to Deoxys-I).

Protected Counter Sums. In [5], Bernstein defined the *protected counter sum* construction, which uses a pseudorandom function $f : \{0,1\}^{d+c} \to \{0,1\}^c$ to build a pseudorandom function with message space of at most $2^c - 1$ blocks of length d:

$$f'(m_1, \ldots, m_\ell) = f\left(0 \| f(1 \| m_1) \oplus \ldots \oplus f(\ell \| m_\ell)\right) \ .$$

The quantum linearization attack essentially shows that this construction, while classically sound, does not yield a quantum-secure pseudorandom function (even if f itself is a qPRF).

4.2 Attacks on BBB MACs

We consider here a variant of the previous construction typically used to design Beyond Birthday MACs. We focus on deterministic MACs, but as before, the same forgery attacks apply if IVs are used in the final processing of the tag.

In the most generic setting, the input $x_1, \ldots, x_m$ is processed with a TBC $\widetilde{E}_{K,i}$, then combined in two different ways:

$$(x_1, \ldots, x_m) \mapsto f_K\left(\bigoplus_i \widetilde{E}_{K,i}(x_i), \bigoplus_i 2^i \widetilde{E}_{K,i}(x_i)\right) .$$

Here f_K is a function whose details are insignificant for our attack.

A similar observation as above applies. By calling the MAC in superposition with messages of the form $x = b_1||0, \ldots, b_m||0$, we will obtain a periodic function. Indeed, there are now two matrices M_m and M'_m with n rows and m columns, and two column vectors C, C' such that:

$$F(x) = F(b_1, \ldots, b_m) := f_K\left(M_m \times \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} \oplus C, M'_m \times \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} \oplus C'\right) ,$$

where the columns of M_m correspond to $\widetilde{E}_{K,i}(0) \oplus \widetilde{E}_{K,i}(1)$ and the columns of M'_m correspond to $2^i(\widetilde{E}_{K,i}(0) \oplus \widetilde{E}_{K,i}(1))$. Then, as soon as $m \geq 2n+1$, the matrix: $\begin{pmatrix} M_m \\ M'_m \end{pmatrix}$ has $2n$ rows and at least $2n+1$ columns, and so, it has a non-trivial kernel. There exists a non-zero vector α such that

$$M_m\alpha = M'_m\alpha = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix} .$$

This α is a boolean period of F, for which $MAC(x \oplus \alpha) = MAC(x)$. Again, the further we increase m, the bigger the subspace of periods will become. This whole space can be recovered using Brassard and Høyer's extension of Simon's algorithm [20] in polynomial time.

Related works. In [30], Guo, Wang, Hu and Ye used combinations of Simon's algorithm and Grover's algorithm to design forgery attacks on many BBB MACs, in the Q2 setting. With this technique, they found two things. First, state-recovery attacks of complexity $\widetilde{\mathcal{O}}\left(2^{n/2}\right)$ where n is the block size of the underlying block cipher, and the internal state is $2n$ bits in total. This comes from the fact that the same input blocks are processed in two branches separately. The standard use of Simon's algorithm, where a controlled message block x is XORed to an uncontrolled value, allows only to recover this value in one of the branches. The n bits on the other branch have to be guessed with a Grover search, and so, the attack is a Grover-meets-Simon [46] instance. And next, partial key-recovery attacks for parallelizable MACs, of complexity $\mathcal{O}\left(2^{k/2}\right)$, where k is the partial

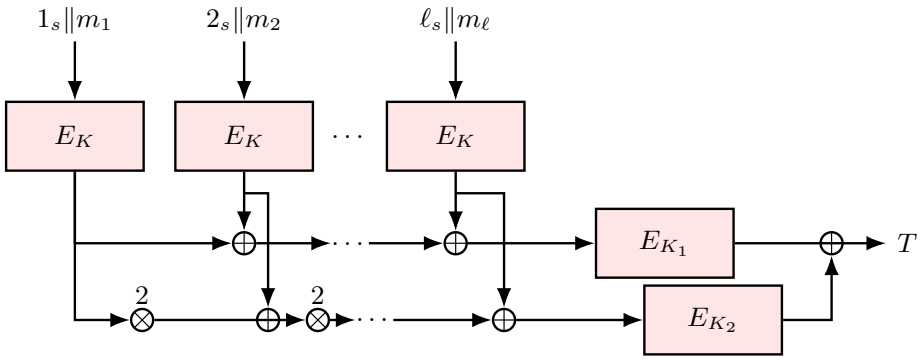


Fig. 3. LightMAC+ with three keys K, K_1, K_2.

key size (the total key size ranges from $3k$ to $5k$). They consist in guessing part of the key and breaking the MAC by using a symmetry of the branches. To these attacks correspond classical partial key-recoveries of complexity $\mathcal{O}\left(2^k\right)$.

Our attack has completely different requirements and offers different results. We need longer messages (of roughly $2n$ blocks in this setting), but when it applies, the complexity is always polynomial. Note that there are constructions targeted by Guo *et al.*, such as SUM-ECBC, that we cannot attack since the blocks are processed sequentially and not linearly in parallel as we require.

LightMAC+. LightMAC+ [54], as its name suggests, is a BBB extension of LightMAC.

As shown in Figure 3, it processes ℓ message blocks $m_1, \ldots, m_\ell$ as follows:

$$\begin{aligned}\mathsf{LightMAC+}(m_1, \ldots, m_\ell) = E_{K_1}(E_K(1_s\|m_1) \oplus \ldots \oplus E_K(\ell_s\|m_\ell)) \\ \oplus E_{K_2}(2^{\ell-1} \odot E_K(1_s\|m_1) \oplus \ldots \oplus 2^0 \odot E_K(\ell_s\|m_\ell)) \ ,\end{aligned}$$

where the multiplications are done in the finite field $\mathbb{F}_{2^n}$. This falls into our framework and is thus forgeable in quadratic time (about $2n$ blocks are required to embed a vector space in both branches, and this can then be recovered in a linear number of queries).

PMAC+. It is a double-block hash-then-sum construction similar to LightMAC+, which also falls into our framework. In full generality, there are three keys K_1, K_2, K_3. The message blocks $m_1, \ldots, m_\ell$ are processed as follows: $y_i = m_i \oplus 2^i \odot E_{K_1}(0) \oplus 2^{2i} \odot E_{K_1}(1)$ and then:

$$\begin{aligned}\mathsf{PMAC+}(m_1, \ldots, m_\ell) = E_{K_2}\left(E_{K_1}(y_1) \oplus \ldots E_{K_1}(y_\ell)\right) \\ \oplus E_{K_3}\left(2 \odot E_{K_1}(y_1) \oplus \ldots 2^\ell \odot E_{K_1}(y_\ell)\right) \ .\end{aligned}$$

The masking by $2^i \odot E_{K_1}(0) \oplus 2^{2i} \odot E_{K_1}(1)$ simply makes the processing of each block different, but this is insignificant for our attack. By recovering a period, we can create forgeries and break PMAC+ as a qPRF.

Note that both LightMAC+ and PMAC+ were classically proven secure up to $2^{3n/4}$ queries [41]. Besides, increasing the number of parallel branches may have

consequences on the bound, but only increases the complexity of our attack by a constant factor. We considered here three-key versions, but of course, the two- and one-key versions [24,25] are similarly broken.

4.3 Other MACs

LAPMAC. LAPMAC was defined in [51]. The definition depends on some parallelization parameter μ. Successive chunks of μ message blocks will be processed in parallel through the block cipher E_K (except the last one), then some tweak function depending on their index in the chunk. The results are XORed and encrypted again, before being XORed to the next chunk of μ message blocks, etc. When $\mu \geq n$, LAPMAC applied to n message blocks becomes similar to LightMAC or PMAC, and there is sufficient parallelization to perform our attack. Whether a variant of the attack applies for smaller values of μ is an interesting question.

ZMAC. ZMAC [37] is a MAC that uses a TBC. It is based on the ZHASH double-block hash construction followed by a finalization function. We can simply focus on the abstraction $\mathbb{ZHASH}$ (see Fig. 5 in [37]):

$$\mathbb{ZHASH}(X_\ell^1, X_r^1, \ldots, X_\ell^l, X_r^l) = \bigoplus_i 2^{l+1-i} \widetilde{E}_K^{i,X_r^i}(X_\ell^i), \bigoplus_i X_r^i \oplus \bigoplus_i \widetilde{E}_K^{i,X_r^i}(X_\ell^i) \ ,$$

where $\widetilde{E}_K^t$ is $\widetilde{E}_K$ called with a tweak t. If we make the tweak inputs constant, then the construction is similar to PMAC+ with a TBC, and different random keyed permutations for each block. Our attack applies as well.

PMAC with a TBC. Naito [53] showed that PMAC+ used with a TBC could achieve full PRF security (up to $\mathcal{O}(2^n)$ queries). In this variant, the message blocks are processed independently with different tweaks. This has no consequence on our attack, which requires only $\mathcal{O}(n)$ queries of about $2n$ blocks each.

5 Attacks on MACs based on Universal Hashing

In this section, we focus on some attacks on MACs based on universal hashing. In particular, we give polynomial-time attacks on PolyMAC [36], GCM-SIV2, and we give a superposition attack on Poly1305 requiring about 32 queries.

5.1 Overview

Universal hash functions were introduced by Carter and Wegman in 1977 [21] in order to build secure MACs, and are now used in many MAC constructions and security proofs. The first proposal by Wegman and Carter was to hash the message and to encrypt the result with a one-time-pad. This defines a MAC with information-theoretic security, but the use of a one-time-pad is impractical, and it was soon suggested to replace it with the output of a PRF, i.e., to replace the one-time-pad by counter-mode encryption. This results in the Wegman-Carter

construction used in GCM and Poly1305-AES: $M \mapsto H_{K_1}(M) \oplus F_{K_2}(N)$ where F is a secure pseudorandom function family, and H an almost-XOR-universal hash function family.

5.2 Universal hash functions and MAC constructions

An almost-XOR-universal hash function family is a family of function H from $\{0,1\}^*$ to $\{0,1\}^n$ indexed by a key $K \in \mathcal{K}$ such that:

$$\forall m \neq m', \forall d \in \{0,1\}^n, \#\{K \in \mathcal{K} : H_K(m) \oplus H_K(m') = d\} \leq \varepsilon \# \mathcal{K}$$

The most widely used universal hash function construction is polynomial hashing. The input message is interpreted as the coefficients of a polynomial in a field $\mathbb{F}$, and the polynomial is evaluated on the hash key:

$$\mathsf{PolyHash}_K : \mathbb{F}^\ell \to \mathbb{F} \qquad m_1, m_2, \ldots m_\ell \mapsto \sum_{i=1}^{\ell} K^i \odot m_i$$

Block cipher-based constructions such as the OCB3 MAC can also be analysed as universal hashing-based, using $\bigoplus_i E_K(A_i \oplus \Delta_i)$ as a universal hash function.

There are many different ways to turn a universal hash function into a MAC:

One-time-MAC: $H_K(M)$. If the universal hash function satisfies extra properties (it must be strongly universal), it can be used directly as a MAC, if a new key is used for each message. This construction is used in ChaCha20-Poly1305, Grain128A and Grain128AEAD [32].

Wegman-Carter: $H_{K_1}(M) \oplus F_{K_2}(N)$. The Wegman-Carter construction is a nonce-based MAC using a universal hash function H and a PRF F. It authenticates several messages using the same key, as long as the nonce N is not repeated (the security is lost as soon as two different messages are authenticated with the same key). This construction is used in GMAC.
More generally, the construction $H(M) \star F(N)$ with $\star$ a group operation and F almost-$\star$-universal is a secure MAC. This construction is used in Poly1305-AES.

Hash-then-PRF: $F_{K_2}(H_{K_1}(M))$. The hash-then-PRF construction builds a deterministic MAC from a universal hash function H and a PRF F.
The PolyMAC construction discussed below follows this design. More generally, security proofs for several block cipher-based MACs consider the MAC as following the hash-then-PRF construction; in particular this is the case of double-block hash-then-sum constructions [24].

WMAC: $F_{K_2}(H_{K_1}(M) \| N)$. WMAC [10] is a generalization of the hash-then-PRF construction using an additional nonce input N to the PRF. This requires a PRF with a larger input, but provides higher security when nonces are unique, without breaking down when they are repeated.

EWCDM: $E_{K_3}(E_{K_2}(N) \oplus N \oplus H_{K_1}(M))$. The *Encrypted Wegman-Carter with Davies-Meyer* construction [23] is an alternative construction offering high security with a nonce with graceful degradation when nonces are repeated. Instead of using a $2n$-bit PRF as in WMAC, it uses two calls to an n-bit block cipher.

5.3 Attacking Wegman-Carter MACs

All MACs following the Wegman-Carter construction are exposed to the attack using Deutsch's algorithm that we presented in Section 3.1. More precisely, an IV-respecting quantum adversary can retrieve the value of $H_{K_1}(M_1) \oplus H_{K_1}(M_2)$ for an arbitrary pair of messages M_1, M_2. He can then repeatedly query the tag of M_1 under new nonces, and produce corresponding valid tags for M_2.

When using the generalization with a group operation $\star$ instead of $\oplus$, this simple attack does not apply. In particular, Poly1305-AES uses a modular addition and cannot be broken with Deutsch's algorithm, but we will show a dedicated attack in Section 5.5, using the fact that it is based on polynomial hashing.

5.4 Attacking Algebraic Universal Hash Functions

We can apply our linearization attack to MACs that reuse the same hash key for several messages, whether deterministic (like hash-then-PRF), or nonce-based (like Wegman-Carter, WMAC, and EWCDM). Indeed, it is enough for us to linearize the function H, and the attack applies regardless of the security of the operations that are computed afterwards, even if they involve a nonce.

Many Universal Hash Functions based on algebraic operations have a strong linear structure. In particular, polynomial hashing is a linear function of the message, making it a natural target for Simon's algorithm (in characteristic 2) or Shor's algorithm (in general). We describe concrete attacks against a few constructions.

PolyMAC. PolyMAC [24] is a double block hash-then-sum construction based on polynomial hashing. The generic construction uses two hashing keys K_1, K_2 and two encryption keys K_3, K_4. For an ℓ-block message $m_1, \ldots, m_\ell$, this gives:

$$\begin{aligned}\mathrm{PolyMAC}(m_1, \ldots, m_\ell) = E_{K_3}\left(K_1 \odot m_\ell \oplus K_1^2 \odot m_{\ell-1} \oplus \ldots \oplus K_1^\ell \odot m_1\right) \\ \oplus E_{K_4}\left(K_2 \odot m_\ell \oplus K_2^2 \odot m_{\ell-1} \oplus \ldots \oplus K_2^\ell \odot m_1\right) \ .\end{aligned}$$

If a single branch is used, then this looks like the GMAC construction [50] (but without a nonce), using polynomial hashing. GMAC was already attacked in [39] due to its similarities with CBC-MAC, and the fact that the nonce did not influence the embedded hidden shift. However, we can use our attack here. By taking ℓ-block message inputs with blocks 0 or 1, we will recover with Simon's algorithm a period $b_1 \cdots b_\ell$ such that:

$$\bigoplus_i b_i K_1^i = 0 \text{ and } \bigoplus_i b_i K_2^i = 0 \ .$$

This immediately allows a forgery attack, but also, we can recover multiple such periods and solve the corresponding equations to recover K_1 and K_2.

PolyMAC with Modular Additions. Interestingly, our attack applies as well when the polynomial hashing does not use XORs, but modular additions (modulo some value M). However, Simon's algorithm has to be replaced by Shor's algorithm. Note that this is specific to polynomial hashing, and does not apply to LightMAC or PMAC-style constructions in general.

We can define:

$$\begin{aligned}\mathsf{PolyMAC+}(m_1,\ldots,m_\ell) = E_{K_3}\left(K_1 \odot m_\ell + K_1^2 \odot m_{\ell-1} + \ldots + K_1^\ell \odot m_1 \bmod M\right)\\ \oplus E_{K_4}\left(K_2 \odot m_\ell + K_2^2 \odot m_{\ell-1} + \ldots + K_2^\ell \odot m_1 \bmod M\right) \ .\end{aligned}$$

In that case, we can remark that there exists periods $a_1,\ldots,a_\ell$ such that:

$$K_1 a_1 + \ldots + K_1^\ell a_\ell \bmod M = 0 \text{ and } K_2 a_1 + \ldots + K_2^\ell a_\ell \bmod M = 0 \ .$$

More precisely, these periods form a lattice in $\mathbb{Z}_M^\ell$, and for all of them, we have:

$$\forall m_1,\ldots,m_\ell, \mathsf{PolyMAC+}(m_1+a_1,\ldots,m_\ell+a_\ell) = \mathsf{PolyMAC+}(m_1,\ldots,m_\ell) \ .$$

Thus, the generalization by Mosca and Ekert [52] of Shor's algorithm allows to retrieve the full lattice of these periods: we can not only forge, but also recover the internal hashing keys.

GCM-SIV2. This is a double-block variant of GCM-SIV defined in [36]. The tag generation combines two independent polynomial hashes (with two keys K_1, K_2) with a keyed-dependent combination function F_K, of which we shall not study the details. This mode is nonce-based. With an empty associated data, the tag is computed as follows:

$$\begin{aligned}\mathsf{GCM\text{-}SIV2-MAC}(N, m_1,\ldots,m_\ell) =\\ F_K\left(N \oplus H_{K_1}(m_1,\ldots,m_\ell), N \oplus H_{K_2}(m_1,\ldots,m_\ell)\right) \ ,\end{aligned}$$

where H_{K_1} and H_{K_2} are polynomial hashes (this would be similar for the tag of an empty message, replacing M by the associated data). Thus, although the MAC is nonce-dependent, it falls into our framework since the periods of the polynomial hashes remain independent of N.

Other algebraic hashing constructions. There are many alternatives to polynomial hashing based on field operations. Several constructions are linear, such as the dot product construction, and Toeplitz hashing [42].[4]

Some other constructions can be linearized using specially crafted messages.

NMH [31].* The NMH* universal hash function is defined as:

$$\mathrm{NMH}^*(M) = \sum (m_{2i} + K_{2i})(m_{2i+1} + K_{2i+1}) \bmod p$$

If we consider messages with blocks with an even index set to arbitrary constants, we obtain a linear function of the odd message blocks. Therefore, Shor's algorithm can break MACs based on this hash function that reuse the hash key.

[4] Grain128A and Grain128AEAD [32] use Toeplitz hashing, but we can only attack them in the nonce-misuse setting because they use the one-time-MAC construction.

BRW Hashing [7]. The BRW universal hash function is based on a class of polynomials that can be evaluated with $\ell/2$ multiplications with ℓ inputs, using a single key. The construction is defined recursively, depending on the input length:

- $BRW_K() = 0$
- $BRW_K(m_1) = m_1$
- $BRW_K(m_1, m_2) = m_1 \odot K + m_2$
- $BRW_K(m_1, m_2, m_3) = (K + m_1) \odot (K^2 + m_2) + m_3$
- $BRW_K(m_1, m_2, \dots m_\ell) = BRW_K(m_1, m_2, \dots m_{t-1}) \odot (K^t + m_t) +$ $BRW_K(m_{t+1}, m_{t+2}, \dots m_\ell)$ with t a power of 2, and $4 \leq t \leq n < 2t$.

For instance, with 8 inputs, we obtain

$$\Big(\big((K+m_1)\odot(K^2+m_2)+m_3\big)\odot(K^4+m_4)+(K+m_5)\odot(K^2+m_6)+m_7\Big)\odot(K^8+m_8)$$

This construction can also be linearized by setting message blocks with an even index set to arbitrary constants.

5.5 Period-Finding against Poly1305

Poly1305 [6] is a polynomial MAC with some specific constraints that force a dedicated analysis. It has already been cryptanalysed in [18], where the authors proposed an attack in 2^{38} time and queries. The authors managed to overcome the specific constraints by leveraging a *hidden shift* structure. The attack we propose here is drastically more efficient, and uses a *hidden period* instead.

Poly1305 uses a hashing key r of 124 bits with at most 106 non-zero bits and a 128-bit cipher key K. The MAC of a message $m_1, \dots, m_\ell$ with the nonce N is computed as:

$$\begin{aligned}\mathsf{Poly1305}(m_1, \dots, m_\ell) = (c_1 r^\ell + c_2 r^{\ell-1} + \dots + c_\ell r^1) \bmod 2^{130} - 5 \\ + AES_K(N) \bmod 2^{128}\ ,\end{aligned}$$

where $c_1, \dots, c_\ell$ are the padded message blocks obtained from the message blocks $m_1, \dots, m_\ell$. When message blocks are full 128-bit blocks, the c_i are simply obtained from the m_i by adding 2^{128}.

Let us assume that we query with two message blocks. We have:

$$\begin{aligned}\mathsf{Poly1305}(m_1, m_2) &= \big((m_1 + 2^{128}) \cdot r^2 + (m_2 + 2^{128}) \cdot r\big) \bmod 2^{130} - 5 \\ &\quad + AES_K(N) \bmod 2^{128} \\ &= \big(((m_1 \cdot r + m_2) \cdot r + C_1) \bmod 2^{130} - 5\big) + C_2 \bmod 2^{128}\ ,\end{aligned}$$

where C_1, C_2 are constants of our query that depend on r, K, N. Since the computation ends with a reduction modulo 2^{128}, which is smaller than $2^{130} - 5$, we must actually use a *compressed* instance of Shor's algorithm [48]. This increases mildly the number of queries, by less than a factor 2.

Two inputs (m_1, m_2) and (m'_1, m'_2) lead to the same tag if

$$m_1 r + m_2 = m'_1 r + m'_2 \bmod 2^{130} - 5$$
$$\Leftrightarrow (m_1 - m'_1)r + (m_2 - m'_2) = 0 \bmod 2^{130} - 5 \ .$$

Hence, the periods of the function $\mathsf{Poly1305}(m_1, m_2)$ are solutions of $m_1 r + m_2 = 0 \bmod 2^{130} - 5$.

As the period is modulo $2^{130} - 5$ but the input is 128-bit long, we cannot do the query expected by Shor's algorithm. Still, the fraction of inputs we can actually query is large enough so that we can still apply Shor's algorithm with a *partial* query, and recover efficiently the period.

The initial query is:

$$\frac{1}{2^{128}} \sum_{m_1, m_2 = 0}^{2^{128}-1} |m_1\rangle \, |m_2\rangle \, |\mathsf{Poly1305}(\mathsf{m_1}, \mathsf{m_2})\rangle$$
$$= \frac{1}{2^{128}} \sum_{m_1, m_2 = 0}^{2^{128}-1} |m_1\rangle \, |m_2\rangle \, |f(m_1 r + m_2)\rangle \ .$$

Here, f is a function that depends on r, K, N. The only relevant point is that it does not depend on m_1, m_2 directly, but only on $m_1 r + m_2$. For simplicity, in the following we assume f is a permutation. We will now apply the QFT over $\mathbb{Z}/(2^{130} - 5)$ on the input registers. We note $p = 2^{130} - 5$. We obtain

$$\frac{1}{p} \frac{1}{2^{128}} \sum_{m_1, m_2 = 0}^{2^{128}-1} \sum_{x, y = 0}^{p-1} \exp\left(2i\pi \frac{x m_1 + y m_2}{p}\right) |x\rangle \, |y\rangle \, |f(m_1 r + m_2)\rangle \ .$$

We can rewrite the state by regrouping components with identical $m_1 r + m_2$:

$$\frac{1}{p} \frac{1}{2^{128}} \sum_{x, y = 0}^{p-1} \sum_{c = 0}^{p-1} \sum_{\substack{m_1, m_2 = 0 \\ m_1 r + m_2 = c}}^{2^{128}-1} \exp\left(2i\pi \frac{x m_1 + y m_2}{p}\right) |x\rangle \, |y\rangle \, |f(c)\rangle$$
$$= \frac{1}{p} \frac{1}{2^{128}} \sum_{x, y = 0}^{p-1} \sum_{c = 0}^{p-1} \sum_{\substack{m_1, m_2 = 0 \\ m_1 r + m_2 = c}}^{2^{128}-1} \exp\left(2i\pi \frac{x m_1 + y(c - m_1 r)}{p}\right) |x\rangle \, |y\rangle \, |f(c)\rangle$$
$$= \frac{1}{p} \frac{1}{2^{128}} \sum_{x, y = 0}^{p-1} \sum_{c = 0}^{p-1} \exp\left(2i\pi \frac{yc}{p}\right) \sum_{\substack{m_1, m_2 = 0 \\ m_1 r + m_2 = c}}^{2^{128}-1} \exp\left(2i\pi \frac{m_1 (x - yr)}{p}\right) |x\rangle \, |y\rangle \, |f(c)\rangle$$

Now, we can compute the probability to measure a nonzero tuple (x, y) with $x = yr$.

As there are $p-1$ such tuples, the overall probability is

$$\left(\frac{1}{p}\frac{1}{2^{128}}\right)^2 (p-1)\sum_{c=0}^{p-1}\left(\sum_{\substack{m_1,m_2=0\\ m_1r+m_2=c}}^{2^{128-1}} 1\right)^2$$
$$= \frac{p-1}{p^2 2^{256}}\sum_{c=0}^{p-1}\left(\#\{0\le m_1,m_2<2^{128} : m_1r+m_2=c\}\right)^2$$

Now, as $x\mapsto x^2$ is a convex function, we can use Jensen's inequality:

$$\sum_{i=1}^{n}\frac{1}{n}\alpha_i^2 \ge \left(\sum_{i=1}^{n}\frac{1}{n}\alpha_i\right)^2 .$$

This allows us to lower bound the previous probability by

$$\frac{p-1}{p^2 2^{256}} p\left(\sum_{c=0}^{p-1}\frac{1}{p}\#\{0\le m_1,m_2<2^{128} : m_1r+m_2=c\}\right)^2$$
$$= \frac{p-1}{p2^{256}}\left(\frac{1}{p}\#\{0\le m_1,m_2<2^{128}\}\right)^2 = \frac{p-1}{p2^{256}}\left(\frac{2^{256}}{p}\right)^2 = \frac{(p-1)2^{256}}{p^3} > \frac{1}{16} .$$

Thus, we measure a tuple $(x,y)\neq(0,0)$ with $x=yr$ with probability at least $1/16$. As $2^{130}-5$ is prime, one such tuple is enough to recover r. Hence, we need at most 16 queries on average to recover r, assuming f is a permutation. Here, as f is a function, we rely on [48] to bound the increase by a factor 2. Note that as we are only a few bits of output short of having a permutation, this is a very loose bound. Overall, the attack will require no more than 32 queries.

6 On Parallelizable Quantum PRFs

Let us take a broader point of view. The deterministic MACs that we attacked in this paper all have common points. Besides allowing inputs of any length (as should be expected of any MAC construction), they • process their input blocks independently; • compute one or more linear functions, *with XORs*, of these processed input blocks; • process the authentication tag from the outputs of these linear functions.

These characteristics are to be expected from any MAC that is: • of average rate one, meaning that there are as many primitive calls as there are blocks; • parallelizable; • having an internal state of size $\mathcal{O}(n)$, independent of the query length. Our attack is easily defeated if the blocks are processed sequentially by calling a compression function, as in the NMAC construction. However, the construction becomes unparallelizable.

It may be possible to obtain a quantum-secure parallelizable qPRF using a tree hashing, where the blocks are placed at the leaves of a binary tree, and each

node is computed by calling a (keyed) compression function on its two children. However, such a construction requires an internal state greater than $\mathcal{O}(n)$, and that increases with the amount of data. Typically to traverse the binary tree, we will need to remember $\mathcal{O}(\log m)$ nodes, where m is the number of leaves.

Open Question. If we stand by the characteristics listed above (efficient, parallelizable, constant internal state size), then it seems that the only solution is to use modular additions in place of XORs in the constructions that we attacked. In that case, our attack does not seem to work anymore, due to the fact that modular additions, contrary to XORs, are not involutive. Thus changing one of the blocks in our n-block queries does not modify involutively the result, which breaks the periodicity property that we used with Simon's algorithm.

This makes this option worth investigating, both from a provable security and a cryptanalysis perspective. Note that the situation is different from most attacks with Simon's algorithm, where the replacement of XORs by + changes the attack complexity from polynomial to subexponential (see [2,18]). In our case, it is possible that using + allows an exponential security level.

7 Conclusion

In this paper, we introduced a novel way of using quantum period-finding to break parallelizable MAC constructions in the superposition query model, breaking most of them in this setting. In full generality, our attack makes use of multiple blocks to embed a hidden period, a surprisingly simple idea that might have other applications. We gave new polynomial-time forgery or partial key-recovery attacks on LightMAC, LightMAC+, PolyMAC, Poly1305, GCM-SIV2, Deoxys, ZMAC, PMAC_TBC3k. Our attack is not mitigated by the use of multiple parallel branches (as in double-block hash-then-sum MACs). It can be prevented for IV-based MACs if the non-reused IV intervenes in the processing of all message blocks (as done in [9]).

These results show that we cannot obtain a parallelizable quantum-secure PRF by processing independently the message blocks, XORing the results, and then hashing the output. If modular additions are used instead of XORs, our attack does not apply anymore (except on polynomial hashing, which has a simpler structure). Overcoming this limitation, or on the contrary, proving the security of such a PRF, is an interesting open question.

Acknowledgments This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 714294 - acronym QUASYModo). A.S. is supported by ERC-ADG-ALGSTRONGCRYPTO (project 740972).

References

1. Alagic, G., Majenz, C., Russell, A., Song, F.: Quantum-access-secure message authentication via blind-unforgeability. In: EUROCRYPT (3). Lecture Notes in Computer Science, vol. 12107, pp. 788–817. Springer (2020)

2. Alagic, G., Russell, A.: Quantum-secure symmetric-key cryptography based on hidden shifts. In: EUROCRYPT (3). Lecture Notes in Computer Science, vol. 10212, pp. 65–93 (2017)
3. Anand, M.V., Targhi, E.E., Tabia, G.N., Unruh, D.: Post-quantum security of the CBC, CFB, OFB, CTR, and XTS modes of operation. In: PQCrypto. Lecture Notes in Computer Science, vol. 9606, pp. 44–63. Springer (2016)
4. Bellare, M., Guérin, R., Rogaway, P.: XOR MACs: New methods for message authentication using finite pseudorandom functions. In: CRYPTO. Lecture Notes in Computer Science, vol. 963, pp. 15–28. Springer (1995)
5. Bernstein, D.J.: How to stretch random functions: The security of protected counter sums. J. Cryptol. 12(3), 185–192 (1999)
6. Bernstein, D.J.: The Poly1305-AES message-authentication code. In: FSE. Lecture Notes in Computer Science, vol. 3557, pp. 32–49. Springer (2005)
7. Bernstein, D.J.: Polynomial evaluation and message authentication (2007), http://cr.yp.to/papers.html#pema
8. Bernstein, E., Vazirani, U.V.: Quantum complexity theory. SIAM J. Comput. 26(5), 1411–1473 (1997)
9. Bhaumik, R., Bonnetain, X., Chailloux, A., Leurent, G., Naya-Plasencia, M., Schrottenloher, A., Seurin, Y.: QCB: efficient quantum-secure authenticated encryption. IACR Cryptol. ePrint Arch. 2020, 1304 (2020)
10. Black, J., Cochran, M.: MAC reforgeability. In: FSE. Lecture Notes in Computer Science, vol. 5665, pp. 345–362. Springer (2009)
11. Black, J., Rogaway, P.: CBC MACs for arbitrary-length messages: The three-key constructions. In: CRYPTO. LNCS, vol. 1880, pp. 197–215. Springer (2000)
12. Black, J., Rogaway, P.: A block-cipher mode of operation for parallelizable message authentication. In: Knudsen, L.R. (ed.) EUROCRYPT. LNCS, vol. 2332, pp. 384–397. Springer (2002)
13. Boneh, D., Zhandry, M.: Quantum-secure message authentication codes. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 7881, pp. 592–608. Springer (2013)
14. Boneh, D., Zhandry, M.: Secure signatures and chosen ciphertext security in a quantum computing world. In: CRYPTO (2). Lecture Notes in Computer Science, vol. 8043, pp. 361–379. Springer (2013)
15. Bonnetain, X.: Quantum key-recovery on full AEZ. In: SAC. Lecture Notes in Computer Science, vol. 10719, pp. 394–406. Springer (2017)
16. Bonnetain, X.: Tight bounds for Simon's algorithm. IACR Cryptol. ePrint Arch. 2020, 919 (2020)
17. Bonnetain, X., Hosoyamada, A., Naya-Plasencia, M., Sasaki, Y., Schrottenloher, A.: Quantum attacks without superposition queries: The offline Simon's algorithm. In: ASIACRYPT (1). Lecture Notes in Computer Science, vol. 11921, pp. 552–583. Springer (2019)
18. Bonnetain, X., Naya-Plasencia, M.: Hidden shift quantum cryptanalysis and implications. In: ASIACRYPT (1). Lecture Notes in Computer Science, vol. 11272, pp. 560–592. Springer (2018)
19. Bonnetain, X., Naya-Plasencia, M., Schrottenloher, A.: On quantum slide attacks. In: SAC. Lecture Notes in Computer Science, vol. 11959, pp. 492–519. Springer (2019)
20. Brassard, G., Høyer, P.: An exact quantum polynomial-time algorithm for Simon's problem. In: ISTCS. pp. 12–23. IEEE Computer Society (1997)
21. Carter, L., Wegman, M.N.: Universal classes of hash functions (extended abstract). In: STOC. pp. 106–112. ACM (1977)

22. Cid, C., Hosoyamada, A., Liu, Y., Sim, S.M.: Quantum cryptanalysis on contracting feistel structures and observation on related-key settings. In: INDOCRYPT. Lecture Notes in Computer Science, vol. 12578, pp. 373–394. Springer (2020)
23. Cogliati, B., Seurin, Y.: EWCDM: an efficient, beyond-birthday secure, nonce-misuse resistant MAC. In: CRYPTO (1). Lecture Notes in Computer Science, vol. 9814, pp. 121–149. Springer (2016)
24. Datta, N., Dutta, A., Nandi, M., Paul, G.: Double-block hash-then-sum: A paradigm for constructing BBB secure PRF. IACR Trans. Symmetric Cryptol. 2018(3), 36–92 (2018)
25. Datta, N., Dutta, A., Nandi, M., Paul, G., Zhang, L.: Single key variant of PMAC_Plus. IACR Trans. Symmetric Cryptol. 2017(4), 268–305 (2017)
26. Deutsch, D.: Quantum theory, the church–turing principle and the universal quantum computer. In: Proceedings of the Royal Society London A. vol. 400, pp. 97—-117. Springer (1985)
27. Deutsch, D., Jozsa, R.: Rapid solution of problems by quantum computation. Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences 439(1907), 553–558 (1992)
28. Dong, X., Dong, B., Wang, X.: Quantum attacks on some feistel block ciphers. Des. Codes Cryptogr. 88(6), 1179–1203 (2020)
29. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: STOC. pp. 212–219. ACM (1996)
30. Guo, T., Wang, P., Hu, L., Ye, D.: Attacks on beyond-birthday-bound macs in the quantum setting. In: PQCrypto. Lecture Notes in Computer Science, vol. 12841, pp. 421–441. Springer (2021)
31. Halevi, S., Krawczyk, H.: MMH: software message authentication in the gbit/second rates. In: FSE. Lecture Notes in Computer Science, vol. 1267, pp. 172–189. Springer (1997)
32. Hell, M., Johansson, T., Meier, W., Sönnerup, J., Yoshida, H.: Grain-128 AEAD a lightweight AEAD streamcipher. Submission to NIST-LWC (2nd Round) (2019)
33. Hosoyamada, A., Sasaki, Y.: Quantum demiric-selçuk meet-in-the-middle attacks: Applications to 6-round generic feistel constructions. In: SCN. Lecture Notes in Computer Science, vol. 11035, pp. 386–403. Springer (2018)
34. Ito, G., Hosoyamada, A., Matsumoto, R., Sasaki, Y., Iwata, T.: Quantum chosen-ciphertext attacks against feistel ciphers. In: CT-RSA. Lecture Notes in Computer Science, vol. 11405, pp. 391–411. Springer (2019)
35. Iwata, T., Kurosawa, K.: OMAC: one-key CBC MAC. In: FSE. LNCS, vol. 2887, pp. 129–153. Springer (2003)
36. Iwata, T., Minematsu, K.: Stronger security variants of GCM-SIV. IACR Trans. Symmetric Cryptol. 2016(1), 134–157 (2016)
37. Iwata, T., Minematsu, K., Peyrin, T., Seurin, Y.: ZMAC: A fast tweakable block cipher mode for highly secure message authentication. In: CRYPTO (3). Lecture Notes in Computer Science, vol. 10403, pp. 34–65. Springer (2017)
38. Jean, J., Nikolic, I., Peyrin, T., Seurin, Y.: Deoxys v1. 41. Submitted to CAESAR (2016)
39. Kaplan, M., Leurent, G., Leverrier, A., Naya-Plasencia, M.: Breaking symmetric cryptosystems using quantum period finding. In: CRYPTO (2). Lecture Notes in Computer Science, vol. 9815, pp. 207–237. Springer (2016)
40. Kaplan, M., Leurent, G., Leverrier, A., Naya-Plasencia, M.: Quantum differential and linear cryptanalysis. IACR Trans. Symmetric Cryptol. 2016(1), 71–94 (2016)

41. Kim, S., Lee, B., Lee, J.: Tight security bounds for double-block hash-then-sum MACs. In: EUROCRYPT (1). Lecture Notes in Computer Science, vol. 12105, pp. 435–465. Springer (2020)
42. Krawczyk, H.: New hash functions for message authentication. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 921, pp. 301–310. Springer (1995)
43. Krovetz, T., Rogaway, P.: The software performance of authenticated-encryption modes. In: FSE. Lecture Notes in Computer Science, vol. 6733, pp. 306–327. Springer (2011)
44. Kuwakado, H., Morii, M.: Quantum distinguisher between the 3-round feistel cipher and the random permutation. In: ISIT. pp. 2682–2685. IEEE (2010)
45. Kuwakado, H., Morii, M.: Security on the quantum-type even-mansour cipher. In: ISITA. pp. 312–316. IEEE (2012)
46. Leander, G., May, A.: Grover meets Simon - quantumly attacking the FX-construction. In: ASIACRYPT (2). Lecture Notes in Computer Science, vol. 10625, pp. 161–178. Springer (2017)
47. Luykx, A., Preneel, B., Tischhauser, E., Yasuda, K.: A MAC mode for lightweight block ciphers. In: FSE. Lecture Notes in Computer Science, vol. 9783, pp. 43–59. Springer (2016)
48. May, A., Schlieper, L.: Quantum period finding is compression robust. CoRR abs/1905.10074 (2019)
49. McGrew, D.A., Viega, J.: The security and performance of the Galois/Counter Mode (GCM) of operation. In: INDOCRYPT. LNCS, vol. 3348, pp. 343–355. Springer (2004)
50. McGrew, D.A., Viega, J.: The security and performance of the galois/counter mode (GCM) of operation. In: INDOCRYPT. Lecture Notes in Computer Science, vol. 3348, pp. 343–355. Springer (2004)
51. Minematsu, K.: A lightweight alternative to PMAC. In: SAC. Lecture Notes in Computer Science, vol. 11959, pp. 393–417. Springer (2019)
52. Mosca, M., Ekert, A.: The hidden subgroup problem and eigenvalue estimation on a quantum computer. In: QCQC. Lecture Notes in Computer Science, vol. 1509, pp. 174–188. Springer (1998)
53. Naito, Y.: Full PRF-secure message authentication code based on tweakable block cipher. In: ProvSec. Lecture Notes in Computer Science, vol. 9451, pp. 167–182. Springer (2015)
54. Naito, Y.: Blockcipher-based MACs: beyond the birthday bound without message length. In: ASIACRYPT (3). Lecture Notes in Computer Science, vol. 10626, pp. 446–470. Springer (2017)
55. Nielsen, M.A., Chuang, I.: Quantum computation and quantum information (2002)
56. NIST: Submission requirements and evaluation criteria for the post-quantum cryptography standardization process (2016), https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf
57. Rogaway, P.: Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In: ASIACRYPT. Lecture Notes in Computer Science, vol. 3329, pp. 16–31. Springer (2004)
58. Santoli, T., Schaffner, C.: Using Simon's algorithm to attack symmetric-key cryptographic primitives. Quantum Inf. Comput. 17(1&2), 65–78 (2017)
59. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: FOCS. pp. 124–134. IEEE Computer Society (1994)
60. Simon, D.R.: On the power of quantum computation. SIAM J. Comput. 26(5), 1474–1483 (1997)

61. Song, F., Yun, A.: Quantum security of NMAC and related constructions - PRF domain extension against quantum attacks. In: CRYPTO (2). Lecture Notes in Computer Science, vol. 10402, pp. 283–309. Springer (2017)
62. Yasuda, K.: The sum of CBC MACs is a secure PRF. In: CT-RSA. Lecture Notes in Computer Science, vol. 5985, pp. 366–381. Springer (2010)
63. Zhang, L., Wu, W., Sui, H., Wang, P.: 3kf9: Enhancing 3GPP-MAC beyond the birthday bound. In: ASIACRYPT. Lecture Notes in Computer Science, vol. 7658, pp. 296–312. Springer (2012)