# FPGA implementations of SPRING
## And their Countermeasures against Side-Channel Attacks

Hai Brenner[1], Lubos Gaspar[2], Gaëtan Leurent[3],
Alon Rosen[1], François-Xavier Standaert[2]

[1] Interdisciplinary Center, Herzliya, Israel.
[2] ICTEAM/ELEN/Crypto Group, Université catholique de Louvain, Belgium.
[3] Inria, EPI SECRET, Rocquencourt, France.

**Abstract.** SPRING is a family of pseudo-random functions that aims to combine the guarantees of security reductions with good performance on a variety of platforms. Preliminary software implementations for small-parameter instantiations of SPRING were proposed at FSE 2014, and have been demonstrated to reach throughputs within small factors of those of AES. In this paper, we complement these results and investigate the hardware design space of these types of primitives.
Our first (pragmatic) contribution is the first FPGA implementation of SPRING in a counter-like mode. We show that the "rounded product" operations in our design can be computed efficiently, reaching throughputs in the hundreds of megabits/second range within only 4% of the resources of a modern (Xilinx Virtex-6) reconfigurable device. Our second (more prospective) contribution is to discuss the properties of SPRING hardware implementations for side-channel resistance. We show that a part of the design can be very efficiently masked (with linear overhead), while another part implies quadratic overhead due to non-linear operations (similarly to what is usually observed, e.g., for block ciphers). Yet, we argue that for this second part of the design, resistance against "simple power analysis" may be sufficient to obtain concrete implementation security. We suggest ways to reach this goal very efficiently, via shuffling. We believe that such hybrid implementations, where each part of the design is protected with adequate solutions, is a promising topic for further investigation.

## 1 Introduction

The quest for secure and efficiently implemented primitives is an ongoing process in cryptography. In the symmetric setting, recent research has led to the development of many standard and lightweight block ciphers. As recently surveyed during the "Crypto for 2020" workshop, current design approaches have been quite successful in optimizing these primitives for various performance metrics, which has led to their deployment in numerous applications [29,31]. Yet, and although the problem of symmetric encryption may sound solved, at least two important (apparently unrelated) problems remain open.

First, from a theoretical point-of-view, there is still a large gap between the formalism used to argue about security in symmetric cryptography (mainly based on cryptanalysis) and the one in asymmetric cryptography (which widely relies on security reductions). While we do not wish to make statements about which approach should currently be privileged and for what application, we believe that any attempt at closing the gap between these two approaches is interesting, since there seems to be no contradiction between implementation efficiency and security reductions from well-understood problems. Note that closing such a gap can naturally benefit both from better security analysis tools for already deployed constructions (as followed by the recent line of research about key alternating ciphers [2,5,6,17,18]) and from "provably secure" constructions leading to efficient implementations, as we pursue in this paper.

Secondly, although block ciphers that perform well on various types of platforms are now mainstream, the problem of securing their implementations against physical (e.g. side-channel) attacks is still quite open. Indeed, the performance overhead caused by standard countermeasures against such attacks (like masking [11,14,27,28] or shuffling [15,33]) are still significant, and the physical assumptions for these countermeasures to provide the expected security improvements are sometimes hard to achieve. (See for example the discussions in [8,22,25].)

Interestingly, and despite looking disconnected, these two problems share a number of (intuitive) similarities, and progresses with respect to one of them could be a source of improvement for the other one. The main reason for this intuition is that one of the main elements that makes asymmetric cryptographic primitives easier to prove by reduction is their more elaborated mathematical structure. But mathematical structure (in particular, certain types of homomorphisms) is exactly what sometimes makes the protection of asymmetric implementations easier, at least from a conceptual point of view [7,16]. As a result, one can generally expect that, as the physical security level of implementations increases, the performance gap between symmetric and asymmetric primitives vanishes. Two recent examples illustrate this hope, namely the masked implementation of the Lapin protocol in [10] and the leakage-resilient MAC in [23]. Unfortunately, both works have some limitations. In the first case, the execution of Lapin requires randomness that seem difficult to protect against side-channel attacks (and was excluded from the physical security analysis so far). In the second case, the MAC relies on quite expensive pairing operations which implies (constant but significant) overhead that dominate the implementation cost in current technologies.

In this paper, we follow this line of work and investigate the implementation properties of a recent Pseudo-Random Function (PRF) candidate called SPRING [3,4]. Based on the "Learning with Rounding" assumption, it enjoys (*i*) compared to the pairings in [23], having underlying operations that can be implemented quite efficiently in software, and (*ii*) compared to Lapin, the advantages of a being deterministic. Besides, and as a PRF, SPRING also corresponds to a stronger and more generic primitive (potentially exploitable for encryption, authentication and hashing). Our contributions are twofold.

We start by describing and evaluating the hardware performance of SPRING within modern FPGAs. For this purpose, we take advantage of its BCH variant described at FSE 2014 and exploit a couple of optimizations, that essentially turn our architecture into a combination of a subset-sum, some Fast Fourier Transforms (FFTs), a rounding step and a BCH code. We show that these operations combine nicely and produce overall performance that is sufficient for a wide range of applications – though still substantially lower than that of AES.

Next, and as part of our motivation relates to physically protected implementation, we also study the extent to which countermeasures against side-channel attacks can be efficiently implemented for SPRING, leading to contrasted conclusions. First, we show that the subset-sum part of our architecture can be masked just as efficiently as Lapin (i.e., independently for each share, with linear overhead). Unfortunately (and quite naturally for a PRF candidate), the rest of its operations are non-linear with respect to the masking scheme, and imply more significant overhead. In this context, we investigate two possible solutions. On the one hand, we estimate the cost of a fully masked implementation, for which the performance is (asymptotically) similar to that of AES. On the other hand, we analyze the cost of a hybrid architecture, where only part of the design is masked and the rest is protected with other means (shuffling, typically). We informally argue about the relevance of this proposal in two directions. First, we observe that standard Differential Power Analysis (DPA) is not possible after the subset-sum operation, because the intermediate computation depends on all the key bits from this point on (so it cannot be enumerated anymore). Secondly, we observe that unmasking before the rounding step will be (theoretically) secure if the leakage function is "rounding" the intermediate values in an appropriate way. In both cases, these arguments suggest that security against Simple Power Analysis (SPA) are sufficient for this part of the design, and we evaluate the cost of a candidate implementation based on this principle.

Overall, our results confirm that SPRING is an interesting family of PRFs for general applications. We also believe that the hybrid countermeasure strategy, that we suggest, is a promising alternative to a secure and efficient implementation, and it leads to interesting open problems. To some extent, it can be viewed as an instantiation of the fresh re-keying scheme in [24], which also combines a DPA-secure linear part with a SPA-secure non-linear one (so connections with such schemes would be interesting to formalize). They also question the families of rounding functions that lead to side-channel resistant PRFs with partial masking, and whether these functions can be implemented physically by a leakage function (possibly as an engineering constraint).

The rest of the paper is structured as follows. Section 2 contains the specifications of SPRING, Section 3 describes its FPGA implementation, and Section 4 discusses its side-channel resistance.

## 2 SPRING specifications

### 2.1 The SPRING Family of Pseudo-Random Functions

One of the main constructions in [4] is a class of PRF candidates called SPRING which is short for "**s**ubset-**p**roduct with **r**ounding over a **ring**." Let $n$ be a power of two, and let $R$ denote the polynomial ring $R := \mathbb{Z}[X]/(X^n + 1)$, which is known as the $2n$th *cyclotomic ring*.[1] For a positive integer $q$, let $R_q$ denote the quotient ring:

$$R_q := R/qR = \mathbb{Z}_q[X]/(X^n + 1),$$

i.e., the ring of polynomials in $X$ with coefficients in $\mathbb{Z}_q$, where addition and multiplication are modulo both $X^n + 1$ and $q$. (For ring elements $r(X)$ in $R$ or $R_q$, the indeterminate $X$ is usually suppressed.) Let $R_q^*$ denote the multiplicative group of units (invertible elements) in $R_q$.

For a positive integer $k$, the SPRING family is the set of functions $F_{a,\boldsymbol{s}} : \{0,1\}^k \to \{0,1\}^m$ indexed by a unit $a \in R_q^*$ and a vector $\boldsymbol{s} = (s_1, \ldots, s_k) \in (R_q^*)^k$ of units. The function is defined as the "rounded subset-product":

$$F_{\boldsymbol{s}}(x_1, \ldots, x_k) := S\left(a \cdot \prod_{i=1}^{k} s_i^{x_i}\right), \tag{1}$$

where $S\colon R_q \to \{0,1\}^m$ for some $m \leq n$ is an appropriate "rounding" function. For example, BPR considers the floor rounding function that maps each of its input's $n$ coefficients to $\mathbb{Z}_2 = \{0,1\}$, depending on whether the coefficient (in its canonical form in $\mathbb{Z}_{257}$) is smaller than $q/2$ or not.

It is proved in [4] that when $a$ and $s_i$ are drawn from appropriate distributions, and $q$ is sufficiently large, the above function family is a secure PRF family, assuming that the "ring learning with errors" (ring-LWE) problem [20] is hard in $R_q$.

## 2.2 Implementation Details: Our Chosen Construction

In the following we describe an optimized FPGA implementation of the SPRING PRF family based on the parameters suggested in [3]. These parameters offer high levels of concrete security against known classes of attacks, and lead to efficient implementations. A discussion and more thorough theoretical analysis of these parameters can be found in [3]. Note that this choice is just one of the possible instantiations of the SPRING family. We use it as a case study to show our hardware implementation and side-channel resistance techniques and to evaluate performance. We also suggest ways to secure the computation from leakage of the key by methods of masking (which seem to match the homomorphic characteristics in the subset-sum part of the SPRING PRF quite elegantly) and alternative countermeasures.

Aiming to design practical functions, the SPRING family can be instantiated with relatively small moduli $q$, rather than the large ones required by the theoretical security reductions in [4]. This allows following the same basic construction paradigm as in [4], while taking advantage of the fast integer arithmetic

[1] It is the $2n$th cyclotomic ring because the complex roots of $X^n + 1$ are all the $2n$th primitive roots of unity. The BPR functions can be defined over other cyclotomic rings as well, but in this work we restrict to powers of two for simplicity and efficiency.

operations. In this paper we follow suit the parameters chosen in [3]:

$$n = 128, \quad q = 257, \quad k = 64,$$

which yields attractive performance, and allows for a comfortable margin of security. The choice of modulus $q = 257$ is akin to the one made in SWIFFT, for a practical instantiation of a theoretically sound lattice-based collision-resistant hash function [19]. Also as in SWIFFT, our implementations build on Fast Fourier Transform-like algorithms modulo $q = 257$.

Choosing an odd modulus $q = 257$ admits very fast subset-product computations in $R_q^*$ using Fast Fourier Transform-type techniques (as mentioned above). However, because $q$ is odd, any rounding function $\lfloor\cdot\rceil : R_q \to R_2$, applied to individual coefficients separately, has bias $1/q$ on each of the output bits. Since $q$ is rather small, such a bias is easily noticeable. This poses no problem at all if SPRING is used for authentication schemes. Nevertheless, it clearly renders the function insecure as a PRF.

To reduce bias, a post-processing step $G$ is implemented by using dual BCH error-correcting code: $S(b_1, \ldots, b_n) = G(\lfloor b_1, \ldots, b_n \rceil)$, where $\lfloor\cdot\rceil$ is applied point-wise. $G$ multiplies the 128-dimensional, $1/q$-biased bit vector by the $64 \times 128$ generator matrix of a binary (extended) BCH error-correcting code with parameters $[n, m, d] = [128, 64, 22]$, yielding a syndrome with respect to the dual code. This simple and very fast "deterministic extraction" procedure (proposed in [1]) reduces the bias exponentially in the distance $d = 22$ of the code, and yields a 64-dimensional vector that is $2^{-145}$-far from uniform (when applied to a 128-dimensional bit vector of independent $1/q$-biased bits). However, this comes at the cost of outputting $m = 64$ bits instead of $n = 128$, as determined by the rate $m/n$ of the code.

In terms of implementation, generator matrices of BCH codes over $GF(2)$ are preferable, since the rows of the matrix are cyclic shifts of a single row, which facilitates fast and simple implementation. Note that $n$ is a power of 2, and any BCH code over $GF(2)$ is of length $2^t - 1$ for some integer $t$. To make the matrix compatible with an $n$ that is a power of two, the extended-BCH code can be used. This extended code is obtained in a standard way by appending a parity bit to the codewords, and it increases the code distance $d$ by one. Finally note that for these chosen parameters $n = 128, m = 64$, the BCH code with parameters $[127, 64, 21]$ and its extension with parameters $[128, 64, 22]$ have the largest known minimum distance for these specific rates. For more theoretical details regarding the bias after applying $G$, we refer the reader to [3].

## 2.3 Implementation Details: Fast Subset-Product in $R_q$

The core of the SPRING algorithm is the subset-product calculation expressed by Equation 1. Direct multiplication of polynomials of degree $n$ is quite expensive, but the computation can be efficient using properties of the carefully chosen ring $R_{257}$. Following [3], the Chinese Remainder decomposition of this ring as $R_q \cong \mathbb{Z}_q^n$ given in [19] is used. More precisely, a polynomial in $R_{257}$ is uniquely determined by its evaluation on the $n$-th primitive roots of unity. Denote this isomorphism by $\mathcal{F}$:

$$\mathcal{F} : R_{257} \to \mathbb{Z}_{257}^n,\ b \mapsto (b(\omega^{2i+1}))_{i=0}^{n-1},$$

where $\omega$ denotes a $2n$-th primitive root of unity. In particular, the multiplicative group of units $R_q^*$ is the set of polynomials whose $\mathcal{F}$ coefficients are all non-zero. This gives the following negacyclic convolution theorem:

$$\mathcal{F}(a \cdot b) = \mathcal{F}(a) \odot \mathcal{F}(b) \tag{2}$$

where $\cdot$ denotes the polynomial product in $R_{257}$, and $\odot$ is the point-wise multiplication of coefficients in $\mathbb{Z}_{257}$.

Moreover, $\mathcal{F}$ and its inverse can be efficiently implemented using an FFT-like algorithm. More precisely, an FFT of size $n$ over the finite field $\mathbb{Z}_{257}$ evaluates a polynomial at all the $n$-th roots of unity: $\mathrm{FFT} : b \mapsto (b(\omega^{2i}))_{i=0}^{n-1}$ (we use $\omega^2$ as a primitive $n$-th root of unity). If we first multiply the coefficient of $b$ by powers of $\omega$, we have:

$$\begin{aligned} b(\omega^{2i+1}) &= b'(\omega^{2i}), && \text{where } b'_i = b_i \cdot \omega^i. \\ \mathcal{F}(b) &= \mathrm{FFT}(b'), \end{aligned}$$

Finally, the subset-product of SPRING can be written as:

$$a \cdot \prod_{i=1}^{k} s_i^{x_i} = \mathcal{F}^{-1}\left(\mathcal{F}(a) \odot \mathcal{F}(s_1^{x_1}) \odot \cdots \odot \mathcal{F}(s_k^{x_k})\right). \tag{3}$$

The value of the bit $x_i$ determines whether polynomial $s_i$ is involved in the subset-product multiplication: $s_i^{x_i}$ is either $s_i$ or 1. In practice, indices with $x_i = 0$ are just removed from the product. Using the convolution theorem, the polynomial subset-product is computed by multiplying the $\mathcal{F}$ evaluations point-wise and transforming the result back by $\mathcal{F}^{-1}$. The $\mathcal{F}$ evaluations can be computed just once beforehand, and stored instead of the polynomials. Applying $\mathcal{F}$ on the $a$ and $s_i$, we obtain these sequences:

$$\begin{aligned} \mathcal{F}(a_0, a_1, \ldots, a_{n-1}) &= [A_0, A_1, \ldots, A_{n-1}], \\ \mathcal{F}(s_{i,0}, s_{i,1}, \ldots, s_{i,n-1}) &= [S_{i,0}, S_{i,1}, \ldots, S_{i,n-1}], \qquad \forall i \in \{1, \ldots, k\}. \end{aligned}$$

We can further simplify the implementation by storing only the discrete logs of the sequences $A$ and $S_i$ using a suitable generator element $G$ (we have chosen $G = 3$). We denote those sequences as $\widehat{A}$ and $\widehat{S_i}$, and the exponentiation as $\mathcal{E}$:

$$\begin{aligned} [A_0, A_1, \ldots, A_{n-1}] &= \mathcal{E}(\widehat{A}) = \left[G^{\widehat{A_0}}, G^{\widehat{A_1}}, \ldots, G^{\widehat{A_{n-1}}}\right], \\ [S_{i,0}, S_{i,1}, \ldots, S_{i,n-1}] &= \mathcal{E}(\widehat{S_i}) = \left[G^{\widehat{S_{i,0}}}, G^{\widehat{S_{i,1}}}, \ldots, G^{\widehat{S_{i,n-1}}}\right], \qquad \forall i \in \{1, \ldots, k\}. \end{aligned}$$

Note the entries $A_j$ and $S_{i,j}$ are all non-zero because $a$ and the $s_i$'s are units in $R_{257}$. If the key is stored in this form, the subset-product in SPRING can be computed very efficiently:

$$a \cdot \prod_{i=1}^{k} s_i^{x_i} = \mathcal{F}^{-1}\left(\mathcal{E}\left(\widehat{A} + \sum_{x_i=1} \widehat{S}_i\right)\right), \tag{4}$$

where the addition is a point-wise modulo 256.

### 2.4 Implementation Details: Counter and Gray-Code mode

As in [3], we focus on implementing SPRING in a counter-like (CTR) mode. This mode uses Gray code, which is a simple way of ordering the strings in $\{0,1\}^k$ so that successive strings differ in only one position. Then, when running SPRING in counter mode, we store the value $\widehat{B} = \widehat{A} + \sum_{x_i=1} \widehat{S}_i$ and we update this additive state by adding or removing a secret key elements $\widehat{S}_i$. Thus, much of the work across consecutive evaluations is amortized.

More precisely, $\widehat{B}$ is initialized to zero (the Gray code starts with $0^k$). For each iteration, if the next input $x'$ flips the $i$th bit of $x$, then the old subset-product is updated to $\widehat{B}' = \widehat{B} + \widehat{S}_i$ if $x_i = 0$, otherwise $\widehat{B}' = \widehat{B} - \widehat{S}_i$.

### 2.5 Operations in SPRING

Thanks to those optimizations, the SPRING evaluations in CRT mode are now reduced to a few simple operations. In the following description, steps 1–3 compute the subset product $b := a \prod_{i=1}^{n} s_i^{x_i}$, and steps 4–5 perform the rounding function $S(b)$:

1. Update the additive state $\widehat{B}$: $\widehat{B} \leftarrow \widehat{B} \pm \widehat{A_i}$, where $i$ is the flipped bit in the Gray counter
2. Compute the polynomial evaluation of $b$ as $B = [G^{\widehat{B_0}}, G^{\widehat{B_1}}, \ldots, G^{\widehat{B_{n-1}}}]$
3. Interpolate the product by computing $\mathcal{F}^{-1}(B)$
   - Compute $b' = \mathrm{FFT}^{-1}(B)$
   - Deduce $b$ with $b_i = b'_i \cdot \omega^{-i}$
4. Round the coefficient of $b$
5. Apply the BCH code

## 3 FPGA implementation

In this section we discuss our design choices for unprotected SPRING-BCH (later just SPRING). We present a SPRING co-processor implementation and report its area and timing performance.

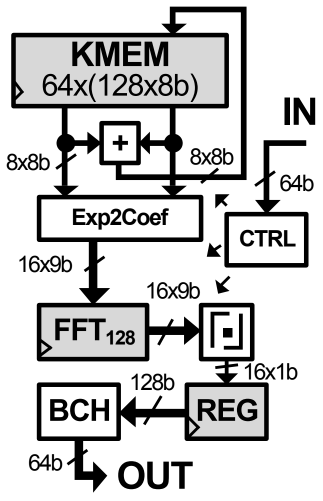

**Fig. 1.** Spring hardware implemenation.

### 3.1 General architecture

Our implementation is depicted in Fig. 1. All arithmetic operations in SPRING are computed modulo $q = 257$ except for the arithmetic operations on exponents which are calculated modulo $q - 1 = 256$. The secret key polynomial FFT coefficients $\widehat{A}$ and $\widehat{S}_i$ (in discrete log form) are stored in true dual port memory KMEM. Each KMEM channel can output 8 exponents (8-bit words) in parallel.

The subset-sum unit (that we detail next) computes addition/subtraction of two polynomials on 8 exponents in parallel, and the results are stored in KMEM. Subsequently, 16 exponents are read from KMEM using both channels and transformed to 9-bit words (representing polynomial evaluations) using table lookups (denoted by Exp2Coef). Next these data are partially processed by the FFT unit (that we also detail next) and stored in its internal registers. This way, in only 8 clock cycles all 128 evaluations are transformed to the polynomial coefficients and stored in the FFT register. When the $\mathrm{FFT}_{128}$ is completed, the resulting 9-bit coefficients are rounded point-wise in 16-coefficient chunks. The floor rounding function replaces each 9 coefficient bits by 1 bit as follows: if a coefficient value is in the range of 0 to 128 the resulting bit equals 0; if a value is in the range of 129 to 256 the resulting bit equals 1. In each clock cycle, 16 rounded 1-bit coefficients are stored in the output data register REG. When all 128 bits are present in REG, they are compressed by the BCH unit to output 64 bits.

### 3.2 Calculation of subset-product

We follow the description of Section 2.3 to implement the subset-product as a subset-sum of exponents, followed by an exponentiation, and the $\mathcal{F}^{-1}$ transform.

We store the coefficient sequences $\widehat{A}$ and $\widehat{S}_i$ in RAM during FPGA configuration. Thus, only the point-wise subset-sum, exponentiation with $G$ (using lookup tables) and $\mathcal{F}^{-1}$ operations have to be implemented in hardware. Unlike direct coefficient representation using 9-bit words, the range of exponents is between 0 and 255, thus the exponents are represented using only 8-bit words. Interestingly, the subset-sum of such exponents involves reduction by 256 instead of 257, which is implemented by simply ignoring the most significant bit (carry) generated by the addition. Moreover, the $\mathrm{FFT}^{-1}$ differs from FFT by multiplication with the constant $n^{-1}$ which is included in KMEM's initialization data. This way, the $\mathrm{FFT}^{-1}$ unit can be replaced by a more simple FFT unit.

## 3.3 Fast Fourier Transform

The FFT on 128 coefficients[2] in parallel is relatively expensive to implement. Instead, we decompose such an FFT unit to smaller FFT blocks. The $\mathrm{FFT}_{128}$ decomposition is illustrated in the left side of Fig. 2, where the input sequence of 128 coefficients is organized in a $2 \times 64$ (row-major) matrix processed in 16-coefficient chunks. After transposing this matrix, an $\mathrm{FFT}_{64}$ is computed on both columns. The results of the two $\mathrm{FFT}_{64}$ blocks are multiplied by powers of a suitable primitive root of unity $\omega$ (we have selected $\omega = 139$), i.e. $\omega^{i \cdot j}$ for $0 \leq i < 64$ and $0 \leq j < 2$. Note, that first 64 powers (with $j = 0$) of $\omega$ are equal to 1, thus no multiplication is required. Although, multiplication with $\omega$ powers is part of the $\mathrm{FFT}_{128}$ computation, it is placed inside the $\mathrm{FFT}_{64}$ block for convenience. Next, an $\mathrm{FFT}_2$ is computed on each row. Finally, the column-major matrix is transposed back to a row-major one and the resulting matrix represents the result of the $\mathrm{FFT}_{128}$ transform. Note that the transposition steps are not shown in Fig. 2, because they are either implemented as a wire crossings (for free in hardware) or they are pre-computed during KMEM initialization.

**$\mathbf{FFT_{64}}$.** In order to simplify the implementation, an $\mathrm{FFT}_{64}$ block is decomposed even further (see Fig. 2), by organizing the 64 input coefficients in an $8 \times 8$ square matrix. Subsequently, the matrix is transposed (part of KMEM initialization), $\mathrm{FFT}_8$ is computed on all 8 rows sequentially, and the result is multiplied by powers of a suitable primitive root of unity $\Omega = \omega^2$ (for $\omega = 139$ we have selected $\Omega = 139^2 \bmod 257 = 46$), i.e. $\Omega^{i \cdot j}$ for $0 \leq i < 8$ and $0 \leq j < 8$. The multiplication results form a new matrix that is stored in the FFT register. Subsequently, this matrix is transposed and stored back in the register. Eventually, $\mathrm{FFT}_8$ operations (described next) are performed on all 8 rows sequentially. Their result is multiplied with $\omega$ powers (as part of the $\mathrm{FFT}_{128}$ computation) and stored back in the FFT register.

**$\mathbf{FFT_8}$.** The basic building block of $\mathrm{FFT}_{64}$ is $\mathrm{FFT}_8$, which is implemented with combinatorial logic only. $\mathrm{FFT}_8$ is composed of three layers of four $\mathrm{FFT}_2$

[2] Actually, the input to $\mathrm{FFT}^{-1}$ is the polynomial *evaluations* and the output is the *coefficients*, but we refer to both as coefficients for simplicity.

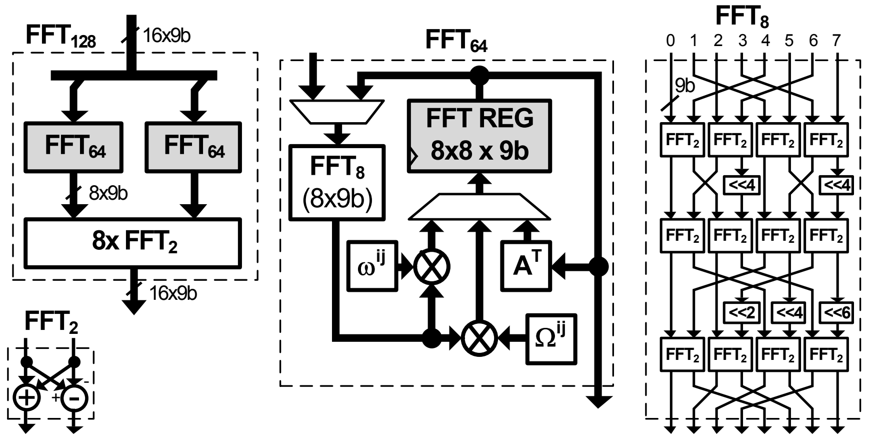

**Fig. 2.** Decomposition of Fast Fourier Transform operating with 128 coefficients.

butterflies, and multiplications by powers of a root of unity. Each butterfly performs one addition and one subtraction. The root of unity $\omega^{16} = 4$ is chosen so that all those constants are powers of two, and can be implemented as shift, as illustrated in Fig. 2. Although the implementation of addition and subtraction is straightforward, the multiplication with a constant could be relatively expensive in $\mathbb{Z}_{257}$. However, $q = 257$ is a Fermat prime number $F_3 = 2^{2^3} + 1 = 257$, thus several interesting properties of arithmetic modulo Fermat numbers can be used to significantly simplify the implementation. Following the description by H. Nussbaumer [26], we introduced an encoding of each 9-bit coefficient $c$ to $C$ as follows:

$$C = (256 - c) \bmod 257. \tag{5}$$

This encoding reduces multiplications inside FFT$_8$ into simple bit rotations around the 9-bit word, with complementation of the overflowing bits.

## 3.4 Cost and performance evaluation

We have synthesized our SPRING co-processor using Xilinx ISE 12.4 for Xilinx Virtex-6 XC6VLX240T FPGAs. The implementation results are summarized in Table 1. KMEM was organized in two 36 kb true dual-port RAMs. As expected, FFT$_{128}$ occupies 76% of the SPRING area. Inside FFT$_{128}$, two blocks of $\Omega^{i\cdot j}$ constant multipliers and one block of $\omega^{i\cdot j}$ constant multipliers utilizes most of the resources (roughly 69% of the FFT$_{128}$). Since these blocks are combinatorial, they are implemented using lookup tables. The two FFT$_8$ blocks constitute 17% of FFT$_{128}$. Next, the two FFT registers storing 1152 bits of the *FFT* state occupy roughly 9% of the FFT$_{128}$ unit. This optimization was achieved using the distributed RAM strategy (only slices are used, no dedicated BRAMs). Other

SPRING parts are relatively small. In total, SPRING occupies 1650 slices which is only 4% of the available FPGA resources.

**Table 1.** Resource usage of the SPRING implementation.

| **Units** | **Slices** | **BRAM(36kb)** |
|---|---|---|
| KMEM | 0 | 2 |
| Subset-sum | 16 | 0 |
| Exp2Coef | 128 | 0 |
| $\mathrm{FFT}_{128}$ total | 1258 | 0 |
| $\rightarrow$ 2x $\mathrm{FFT}_8$ | 210 | 0 |
| $\rightarrow$ 2x FFT REG + transpose | 110 | 0 |
| $\rightarrow$ 2x Mult. $\Omega^{i \cdot j}$ | 496 | 0 |
| $\rightarrow$ 1x Mult. $\omega^{i \cdot j}$ | 378 | 0 |
| $\rightarrow$ 8x $\mathrm{FFT}_2$ | 64 | 0 |
| Rounding + REG | 32 | 0 |
| BCH | 189 | 0 |
| Control logic | 27 | 0 |
| **SPRING - Total** | **1650** | **2** |

As far as speed performance is concerned, SPRING was designed to operate without idle states in the Gray counter mode. Computation starts by sequentially reading rows of 16 exponents from KMEM. These row exponents are first converted to coefficients, each half of them being processed by one of the two $\mathrm{FFT}_8$ blocks, then multiplied with corresponding powers of $\Omega$, and stored in the two FFT registers. All these operations together are executed in only one clock cycle. Subsequently, the same operations are performed on the other 7 rows. Together, all 8 rows are processed in only 8 clock cycles. Next, both FFT registers are transposed in 12 clock cycles. The new 8 rows are processed by the two $\mathrm{FFT}_8$ blocks, multiplied with corresponding powers of $\omega$ and stored back to the two FFT registers in 8 clock cycles. In the subsequent 8 clock cycles, 8 parallel $\mathrm{FFT}_2$'s are computed on 8 rows. Then, all 9-bit coefficients are rounded to 1-bit coefficients and stored in the 128-bit register. The SPRING output is finally obtained by post-processing the combinatorial BCH unit on this register data. In parallel with the last 28 cycles, a new subset-sum is calculated. Since the subset-sum calculation requires 32 clock cycles, 4 extra clock cycles are necessary. This way, one SPRING execution with subset-sum pre-calculation requires 40 clock cycles. Such a result illustrates the usual trade-off between generic software and specialized hardware. In particular, it outperforms the FSE 2014 software implementation on Intel Core i7 Ivy Bridge (392 clock cycles per one encryption) by an approximate factor 10.

Besides, the maximum operating clock frequency for the selected FPGA device was estimated at 91.7 MHz. This seemed to be a good trade-off between speed and implementation size[3]. Assuming this clock frequency and continuous Gray counter mode operation, 2.3 million encryptions can be carried out in 1 second, which corresponds to a 140 megabits/second throughput.

For illustration purposes, we reported the performances of SPRING and a couple of representative algorithms in Table 2. While not directly comparable, they provide insights about the implementation cost of other recent primitives based on the Learning with Errors problem (yet, used for different purposes such as authentication or public-key encryption) and the AES Rijndael (which although based on totally different assumptions, aims at a similar goal as SPRING, namely PRP). As expected, the performance gap between our SPRING design and AES ones is slightly larger than in a software context, essentially because there are more computation units to implement here. Yet, the cost vs. performance trade-off obtained (in the hundreds of megabits/second range for a few %'s of the FPGA resources) is already sufficient for a wide range of applications.

**Table 2.** Comparison of different algorithms implemented on a Virtex 6 FPGA.

| **Algorithm** | **Type** | **Dapath** | **LUT** | **FF** | **BRAM** | **DSP** | **F$_{max}$**[1] | **Cycles** |
|---|---|---|---|---|---|---|---|---|
| SPRING | PRF | 128/144b | 7292 | 294 | 2x 36k | 0 | 91.7 | 40 |
| Lapin [10] | auth. | 128b | 742 | 140 | 6x 36k | 0 | 140.3 | 1332 |
| Comp-LWE [30] | PKE | N/A | 1879 | 1142 | 3x 18k | 1 | 250.0 | 13287 [2] |
| AES-LUT [9] | PRP | 128b | 933 | 399 | 10x 18k | 0 | 674.0 | 11 |
| AES-COMB [9] | PRP | 128b | 2335 | 535 | 0 | 0 | 218.6 | 11 |
| AES-COMB [9] | PRP | 32b | 467 | 976 | 0 | 0 | 315.1 | 58 |

[1] Maximum frequency is denoted in MHz.
[2] Number of clock cycles for encryption only.

## 4 Towards side-channel resistance

We now move to the second part of our investigation, and discuss two possible approaches to side-channel resistance for hardware implementations of SPRING. The first one takes advantage of standard solutions for masking (a.k.a. secret sharing). As we detail next, it implies more significant overhead for certain parts of the computation than others. Motivated by this observation, we then suggest an alternative approach, where only one part of the implementation is masked, and the other one is shuffled. We show that this alternative is very efficient; we argue about its relevance and suggest open problems based on it.

[3] The clock frequency could indeed be higher if the long combinatorial paths were split by pipelining. However, this would result in a substantially larger implementation and latency.

### 4.1 Fully masked design

In this first subsection, we show how to secure the computation of SPRING from leakage of key by means of masking. For this purpose, the key is initially split into random-looking shares that are refreshed before each execution, and the computation is made on each of the shares separately. We next refer to the parts of the hardware which handle the computations on specific shares as *parties*. Let $d$ denote the number of parties. The main intuition behind masking is that no information on the original key can be obtained from the computation of less than $d$ parties.

We start by sketching the different steps of a masked SPRING computation.

1. A synchronization step is required to refresh the key shares that are stored in discrete-log (of FFT evaluations) format. For this purpose, a usual strategy is to add a random sharing of zero to the $d$ additive shares stored in memory.
2. The parties compute their subset-sum locally by using only their share, or they update the subset-sum according to the Gray code counter.
3. The parties change their additive shares of the subset-sum to multiplicative shares of the corresponding subset-product by locally using the Exp2Coef lookup table.
4. A second synchronization step converts the parties' current multiplicative shares into additive shares of the same value. We refer to this unit as MM2AM.
5. The parties locally use the FFT unit on their shares of the computation.
6. A last synchronization step is used for the rounding. The parties now have additive shares of the polynomial subset-product in Equation 1. They compute XOR shares of the rounding bit of the coefficients.
7. The parties apply the (linear) binary BCH transformation on their shares locally.
8. Finally, the parties now have XOR shares of the SPRING evaluation. These bits are XORed to obtain the output.

We now describe the whole secure masked computation more extensively.

The process starts with a standard refreshing of the pre-shared key, which is stored as discrete logs of outputs of the FFT procedure on the polynomials $a, \boldsymbol{s}$. As a result, the parties get numbers whose sum modulo 256 corresponds to an original discrete log in the key. As the first step of the SPRING computation is merely a subset-sum of the key (or an update of it according to the Gray-code counter), each of the parties can compute this subset-sum locally and independently. This unit for each of the parties is identical to the corresponding one in the unprotected SPRING implementation. After this computation, each of the parties has an additive share of the subset-sum (modulo 256).

The next unit in the original computation is the Exp2Coef lookup table. This table is used for putting the discrete logs in $\mathbb{Z}_{256}$ into the exponent to get FFT outputs in $\mathbb{Z}_{257}^*$. We apply exactly the same lookup table locally in the computation of each of the parties in parallel. After passing the subset-sum results in the Exp2Coef lookup table, the parties hold point-wise multiplicative shares (modulo 257) of the corresponding subset-product.

Following, the original SPRING evaluation performs the FFT unit. Notice that it is an additively-linear operation, but currently the parties hold multiplicative masking shares of the computation. We apply a synchronization step to convert the multiplicative shares back to additive ones. The unit responsible for this step is denoted by MM2AM.

For this purpose, we use the technique suggested by Ghodosi *et al.* in [13].[4] This procedure is applied for each entry independently. The MM2AM unit gets in advance random bits and uses them to generate $d^2$ random-looking numbers in $\mathbb{Z}^*_{257}$, such that:

$$\sum_{j=0}^{d-1}\prod_{i=0}^{d-1}\alpha_{i,j} = 1 \pmod{257}.$$

This is easily obtained because we achieve a random number in $\mathbb{Z}^*_{257}$ by adding 1 to an 8-bit random number, and because we can randomize $d^2-1$ numbers and compute $r_{0,0}$ so the equation holds. For each $0 \le i < d$, the MM2AM unit provides party $i$ with the numbers $\{\alpha_{i,j}\}_{j=0,\dots d-1}$ as auxiliary information. This information is independent from the shares so it can be achieved before the computation reaches the Exp2Coef step. Once the parties get the multiplicative shares $\{m_i\}_{i=0,\dots,d-1}$ of the subset-product, each party $i$ sends the value $m_i \cdot \alpha_{i,j}$ to party $j$. Next, each party $j$ computes the multiplication $c_j = \prod_{i=0}^{d-1} m_i \cdot \alpha_{i,j} \pmod{257}$ as its additive share of the entry. Note that:

$$\begin{aligned}\sum_{j=0}^{d-1} c_j &= \sum_{j=0}^{d-1}\left(\prod_{i=0}^{d-1} m_i\right)\cdot\left(\prod_{i=0}^{d-1}\alpha_{i,j}\right),\\ &= \left(\prod_{i=0}^{d-1} m_i\right)\cdot\left(\sum_{j=0}^{d-1}\prod_{i=0}^{d-1}\alpha_{i,j}\right),\\ &= \prod_{i=0}^{d-1} m_i \pmod{257}.\end{aligned}$$

Therefore, the parties now have additive masking shares of this step of the computation. Taking advantage of these additive shares, the implementation can perform the (linear) FFT units locally and in parallel. They are identical to the corresponding units in the original unprotected SPRING implementation.

As a result, the parties have additive shares of the polynomial subset-product. It therefore remains to compute the rounding function. We use a masked rounding unit for this purpose, such that its output for each party is a XOR random share of the actual rounding value. We describe the computation of rounding on a single coefficient. This process is repeated for all $n$ coefficients of the subset-product. The unit deals in advance random shifts $\{t_i\}_{i=0,\dots,d-1} \in \mathbb{Z}^d_{257}$ to the

[4] Other techniques for share conversion exist. The side-channel literature usually refers to [12] for such a task, but the algorithms in this reference are quite sequential. We focus on the approach suggested in [13] which seems easier to parallelize and simpler in our hardware context.

parties. Each party is also provided with a pre-generated random-looking table $TAB_i \in \{0,1\}^{257}$. For each $i \geq 1$ the table is merely a random bit array of size 257. The table of party 0 satisfies:

$$TAB_0[v] = \bigoplus_{i=1}^{d-1} TAB_i[v] \oplus \left\lfloor v - \sum_{i=0}^{d-1} t_i \right\rceil \pmod{257} \qquad \forall v \in \mathbb{Z}_{257}, \tag{6}$$

where $\oplus$ is the XOR operator in $\mathbb{Z}_2$.

Let $v_i$ be the additive share of a single coefficient $v$ kept by party $i$, as computed by the FFT unit. Party $i$ shares $v_i + t_i$ with all the other parties. Next, all parties compute the sum $\sum(v_i + t_i) = v + \sum t_i$, and use $b_i = TAB[v + \sum t_i]$ as their share of the coefficient's rounding output. We note that $\bigoplus_{i=0}^{d-1} b_i$ is indeed the rounded bit of $v$ due to generation of the tables described in Equation 6.

Eventually, as BCH is a linear transformation of the rounding output, the parties compute their shares locally and independently again, using an identical instance of this unit for each party. As a result, all parties have a XOR-share of the PRF evaluation.

**Implementation and cost estimation:** The implementation of the fully masked SPRING is depicted in Fig. 3. It operates with $d$ shares in parallel. Essentially, its datapath is composed of $d$ (modified) datapaths of the unprotected SPRING, where mask refreshing, mask conversion (MM2AM) and masked rounding are added. In order to preserve the same timing as the unprotected SPRING version, random masks are added to the KMEM outputs on-the-fly and the MM2AM and masked rounding units are implemented with combinatorial logic.

The cost estimation of these three new units is summarized in Tab. 3. As can be observed, the impact of mask refreshing is negligible. Moreover, its size increases linearly with the number of shares. The other two units are more expensive. For example, in case of a two-share implementation, the mask conversion MM2AM requires 527 slices, whereas masked rounding requires 1321 slices. Furthermore, the size of MM2AM increases quadratically $d$. This is caused by a substantial increase of the number of multiplications performed inside MM2AM. Interestingly, the masked rounding does not utilize expensive multiplications, and so its size increases linearly with d.

Note that Tab. 3 illustrates the costs to protect one 8-bit sensitive exponent or one 9-bit entry (either a polynomial evaluation before the FFT$^{-1}$ step or a polynomial coefficient after this step). However, the implementation illustrated in Fig. 3 operates on $16d$ exponents/coefficients in parallel. Thus, if implemented, these units would increase SPRING size substantially (just as generally observed for masked implementations of block ciphers).

## 4.2 Partially masked design

In view of the previous estimations, it appears that certain parts of the SPRING design (namely, until the FFT computation) are very easy to mask, while the

**Table 3.** Estimated costs of basic operations (dependent on $d$) necessary for mask refreshing, $d$-share multipl. to additive masking conv. and masked rounding of one 9-bit entry.

| | Basic operations | | | | | Random | Total # of slices | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | ADD | MUL | INV | MUX2 | XOR | bits. | $d=2$ | $d=3$ | $d=4$ | $d=5$ |
| Msk. refresh | $d-2$ | 0 | 0 | 0 | 0 | $8(d-1)$ | 3 | 5 | 6 | 7 |
| MM2AM[13] | $d-2$ | $3d^2-2d$ | $d$ | 0 | 0 | $8(d^2-1)$ | 527 | 1353 | 2551 | 4121 |
| Msk. round | $3d-2$ | 0 | 0 | $256d$ | $d-1$ | $266d-257$ | 1321 | 1409 | 1473 | 1894 |

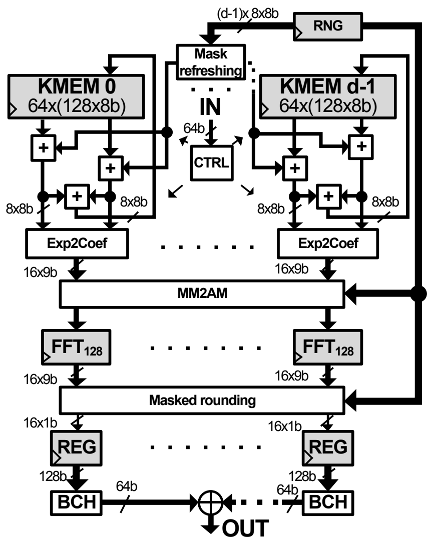

**Fig. 3.** Fully masked implementation.

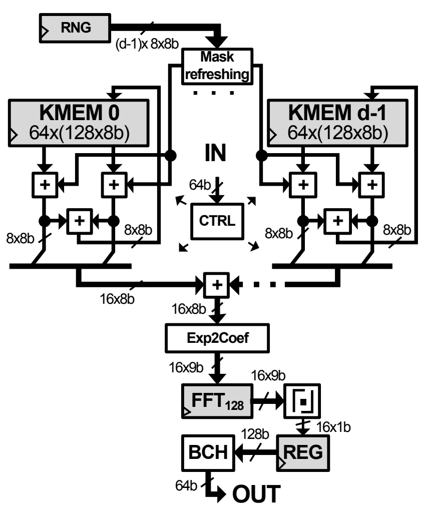

**Fig. 4.** Partially masked implementation.

remaining ones imply the usual (quadratic) overhead of non-linear operations. In this context, an appealing solution from the performance point of view would be to unmask the implementation just before these non-linear steps, as illustrated in Fig. 4. Quite naturally, this raises the question whether the partially masked design becomes insecure at this point. We conclude this paper with two simple arguments in favor of such a hybrid strategy.

First, observe that the standard DPA attacks (defined in [21]), that are at the core of most physical security evaluation procedures nowadays, are inherently limited to the exploitation of the leakages corresponding to operations that can be predicted (i.e., that depend on an enumerable part of the key). In the case of SPRING, such operations only appear at the beginning of the encryption, since diffusion is complete after the subset-sum computation. As a result, it could be sufficient to protect this part of the implementation against DPA, and the remaining ones against (single-trace) SPA. (As mentioned in the introduction, such an idea is reminiscent of fresh re-keying schemes[24].) Interestingly, several cheaper solutions exist for this purpose.

For example, shuffling is a usual countermeasure against SPA. In the case of SPRING, our implementation operates on polynomials of 128 coefficients/exponents, but only 16 are processed in parallel (one row). Therefore, 8 consecutive calculations (per row) are necessary to process the full 128 coefficient/exponent state. Since they are computed independently, these operations can be directly executed according to a random 8-permutation. It represents a total of $8! = 40320$ execution permutations which, combined with the high (algorithmic) noise of our hardware design, should prevent SPA.

Concretely, the eight entry rows are easily shuffled by being read from KMEM in a random order. This involves only small changes in the control logic. Hence its impact on the SPRING implementation size is negligible. The only parts that need modification are the matrix transpose and FFT-REG inside each $\text{FFT}_{64}$. However, this modification only increases the size of both by 24 slices. Furthermore, shuffling can be preserved up to the REG unit, where the rows have to be stored in a correct position. This operation de-shuffles the state for free.

As can be observed, shuffling presents a very powerful cost-efficient countermeasure. For this reason, we have decided to implement this approach in the masked SPRING implementation. Moreover, only the linear part of SPRING (i.e. subset-sum) is protected by the masking countermeasure. This way, linear increase of the number of shares is reflected in the linear increase of implementation size.

To conclude, let us also observe that, depending on the leakage function, unmasking the last part of the design could simply be secure as such. For example, imagine a leakage function that would "round" the intermediate values, just as required by the SPRING specifications. Then, having unmasked data after the FFT computations would not be a problem at all. Quite naturally, actual leakage functions do not round as proposed in [3,4]. Yet, they usually compress the input range to some extent (e.g. with a Hamming weight function). Furthermore, technology-level countermeasures such as dual-rail logic styles can generally be used to modify the shape of the leakage functions [32]. As a result, it is an interesting open problem to find whether there is a common ground for protected implementation of rounded operations, between theoretical requirements and practical engineering constraints.

## Acknowledgements

This work has been funded in part by the ERC project 280141 (acronym CRASH) and in part by the Belgian Cybercrime Center of Excellence for Training Research and Education (B-CCENTRE). F.-X. Standaert is an associate researcher of the Belgian Fund for Scientific Research (FNRS-F.R.S.). Research supported by the ERC under the EU's Seventh Framework Programme (FP/2007-2013) ERC Grant Agreement n. 307952.

## References

1. Alberini, G., Rosen, A.: Efficient Rounding Procedures of Biased Samples. Manuscript (2013)
2. Andreeva, E., Bogdanov, A., Dodis, Y., Mennink, B., Steinberger, J.P.: On the Indifferentiability of Key-Alternating Ciphers. In Canetti, R., Garay, J.A., eds.: CRYPTO (1). Volume 8042 of Lecture Notes in Computer Science., Springer (2013) 531–550
3. Banerjee, A., Brenner, H., Leurent, G., Peikert, C., Rosen, A.: SPRING: Fast Pseudorandom Functions from Rounded Ring Products. to appear in the proceedings of FSE 2014, Lecture Notes in Computer Science, vol xxxx, pp yyy-zzz, London, UK, March 2014.
4. Banerjee, A., Peikert, C., Rosen, A.: Pseudorandom Functions and Lattices. In: EUROCRYPT. (2012) 719–737
5. Bogdanov, A., Knudsen, L.R., Leander, G., Standaert, F.X., Steinberger, J.P., Tischhauser, E.: Key-Alternating Ciphers in a Provable Setting: Encryption Using a Small Number of Public Permutations - (Extended Abstract). In Pointcheval, D., Johansson, T., eds.: EUROCRYPT. Volume 7237 of Lecture Notes in Computer Science., Springer (2012) 45–62
6. Chen, S., Steinberger, J.P.: Tight security bounds for key-alternating ciphers. IACR Cryptology ePrint Archive **2013** (2013) 222
7. Coron, J.S.: Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In Çetin Kaya Koç, Paar, C., eds.: CHES. Volume 1717 of Lecture Notes in Computer Science., Springer (1999) 292–302
8. Coron, J.S., Giraud, C., Prouff, E., Renner, S., Rivain, M., Vadnala, P.K.: Conversion of Security Proofs from One Leakage Model to Another: A New Issue. In Schindler, W., Huss, S.A., eds.: COSADE. Volume 7275 of Lecture Notes in Computer Science., Springer (2012) 69–81
9. Crypto group, UCL, Louvain-la-Neuve, Belgium
10. Gaspar, L., Leurent, G., Standaert, F.X.: Hardware Implementation and Side-Channel Analysis of Lapin. In Benaloh, J., ed.: CT-RSA. Volume 8366 of Lecture Notes in Computer Science., Springer (2014) 206–226
11. Genelle, L., Prouff, E., Quisquater, M.: Thwarting Higher-Order Side Channel Analysis with Additive and Multiplicative Maskings. In Preneel, B., Takagi, T., eds.: CHES. Volume 6917 of Lecture Notes in Computer Science., Springer (2011) 240–255
12. Genelle, L., Prouff, E., Quisquater, M.: Thwarting higher-order side channel analysis with additive and multiplicative maskings. In: Cryptographic Hardware and Embedded Systems–CHES 2011. Springer (2011) 240–255
13. Ghodosi, H., Pieprzyk, J., Steinfeld, R.: Multi-party computation with conversion of secret sharing. Designs, Codes and Cryptography **62**(3) (2012) 259–272 `http://www1.spms.ntu.edu.sg/~ccrg/documents/MPC_SS-sing-2011.pdf`, Feb. 2014.
14. Grosso, V., Standaert, F.X., Faust, S.: Masking vs. Multiparty Computation: How Large Is the Gap for AES? In Bertoni, G., Coron, J.S., eds.: CHES. Volume 8086 of Lecture Notes in Computer Science., Springer (2013) 400–416
15. Herbst, C., Oswald, E., Mangard, S.: An AES Smart Card Implementation Resistant to Power Analysis Attacks. In Zhou, J., Yung, M., Bao, F., eds.: ACNS. Volume 3989 of Lecture Notes in Computer Science. (2006) 239–252
16. Joye, M., Yen, S.M.: The Montgomery Powering Ladder. In Jr., B.S.K., Çetin Kaya Koç, Paar, C., eds.: CHES. Volume 2523 of Lecture Notes in Computer Science., Springer (2002) 291–302

17. Lampe, R., Patarin, J., Seurin, Y.: An Asymptotically Tight Security Analysis of the Iterated Even-Mansour Cipher. [34] 278–295
18. Lampe, R., Seurin, Y.: How to Construct an Ideal Cipher from a Small Set of Public Permutations. In Sako, K., Sarkar, P., eds.: ASIACRYPT (1). Volume 8269 of Lecture Notes in Computer Science., Springer (2013) 444–463
19. Lyubashevsky, V., Micciancio, D., Peikert, C., Rosen, A.: SWIFFT: A Modest Proposal for FFT Hashing. In: FSE. (2008) 54–72
20. Lyubashevsky, V., Peikert, C., Regev, O.: On Ideal Lattices and Learning with Errors over Rings. J. ACM (2013) To appear. Preliminary ver. in Eurocrypt 2010.
21. Mangard, S., Oswald, E., Standaert, F.X.: One for all - all for one: unifying standard differential power analysis attacks. IET Information Security **5**(2) (2011) 100–110
22. Mangard, S., Popp, T., Gammel, B.M.: Side-Channel Leakage of Masked CMOS Gates. In Menezes, A., ed.: CT-RSA. Volume 3376 of Lecture Notes in Computer Science., Springer (2005) 351–365
23. Martin, D., Oswald, E., Stam, M.: A Leakage Resilient MAC. IACR Cryptology ePrint Archive **2013** (2013) 292
24. Medwed, M., Standaert, F.X., Großschädl, J., Regazzoni, F.: Fresh Re-keying: Security against Side-Channel and Fault Attacks for Low-Cost Devices. In Bernstein, D.J., Lange, T., eds.: AFRICACRYPT. Volume 6055 of Lecture Notes in Computer Science., Springer (2010) 279–296
25. Moradi, A., Mischke, O.: How Far Should Theory Be from Practice? - Evaluation of a Countermeasure. In Prouff, E., Schaumont, P., eds.: CHES. Volume 7428 of Lecture Notes in Computer Science., Springer (2012) 92–106
26. Nussbaumer, H.J.: Fast Fourier transform and convolution algorithms. Springer (1982)
27. Rivain, M., Prouff, E.: Provably Secure Higher-Order Masking of AES. In Mangard, S., Standaert, F.X., eds.: CHES. Volume 6225 of Lecture Notes in Computer Science., Springer (2010) 413–427
28. Roche, T., Prouff, E.: Higher-order glitch free implementation of the AES using Secure Multi-Party Computation protocols - Extended version. J. Cryptographic Engineering **2**(2) (2012) 111–127
29. Rombouts, P.: Lightweight Cryptography: Mission Accomplished. Crypto for 2020 – ECRYPT-II Final Event, Tenerife, Spain, January 2013.
30. Roy, S.S., Vercauteren, F., Mentens, N., Chen, D.D., Verbauwhede, I.: Compact Ring-LWE based Cryptoprocessor. Cryptology ePrint Archive, Report 2013/866 (2013) `http://eprint.iacr.org/`.
31. Standaert, F.X.: Future Challenges for Lightweight Cryptography. Crypto for 2020 – ECRYPT-II Final Event, Tenerife, Spain, January 2013.
32. Tiri, K., Verbauwhede, I.: Securing Encryption Algorithms against DPA at the Logic Level: Next Generation Smart Card Technology. In Walter, C.D., Çetin Kaya Koç, Paar, C., eds.: CHES. Volume 2779 of Lecture Notes in Computer Science., Springer (2003) 125–136
33. Veyrat-Charvillon, N., Medwed, M., Kerckhof, S., Standaert, F.X.: Shuffling against Side-Channel Attacks: A Comprehensive Study with Cautionary Note. [34] 740–757
34. Wang, X., Sako, K., eds.: Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings. In Wang, X., Sako, K., eds.: ASIACRYPT. Volume 7658 of Lecture Notes in Computer Science., Springer (2012)