A Language-Based Approach for Improving the Robustness of Network Application Protocol Implementations

Laurent Burgy Laurent Réveillère
Phoenix Group – INRIA / LaBRI
University of Bordeaux, France
{burgy,reveillere}@labri.fr

Julia L. Lawall
University of Copenhagen
Copenhagen, Denmark
julia@diku.dk

Gilles Muller
École des Mines de Nantes
INRIA / LINA, France
Gilles.Muller@emn.fr

Abstract

The secure and robust functioning of a network relies on the defect-free implementation of network applications. As network protocols have become increasingly complex, however, hand-writing network message processing code has become increasingly error-prone.

In this paper, we present a domain-specific language, Zebu, for generating robust and efficient message processing layers. A Zebu specification, based on the notation used in RFCs, describes protocol message formats and related processing constraints. Zebu-based applications are efficient, since message fragments can be specified to be processed on demand. Zebu-based applications are also robust, as the Zebu compiler automatically checks specification consistency and generates parsing stubs that include validation of the message structure. Using a message torture suite in the context of SIP and RTSP, we show that Zebu-generated code is both complete and defect-free.

Keywords: Domain-specific Languages, Message processing, Network protocols.

1. Introduction

In the Internet era, many applications, ranging from instant messaging clients and multimedia players to HTTP servers and proxies, involve processing network protocol messages. A key part of this processing is to parse messages as they are received from the network. As message parsing represents the front line of interaction between the application and the outside world, the correctness of the parser is critical; any bugs can leave the application open to attack [16]. In the context of in-network application such as proxies, where achieving high throughput is essential, parsing must also be efficient.

Implementing a correct and efficient network protocol message parser, however, is a difficult task. One issue is that the Requests for Comments (RFCs) that define these protocols typically specify the message syntax using a variant of BNF. Such a specification amounts to a state machine, which for efficiency is often implemented in an unstructured way using gotos. The resulting code is thus error-prone and difficult to maintain. Another issue is that some kinds of message processing may only use part of the message. For example, a router normally only uses the header fields that describe the message destination, and ignores the header fields that describe properties of the message body [15]. It is thus desirable, for efficiency, to defer the parsing of certain message fragments to when their values are actually used. In this case, complex parsing code may end up scattered throughout the application.

In the programming languages community, parsers have long been constructed using automated parser generators such as `yacc` [9]. Nevertheless, such tools are not suitable for generating parsers for network protocol messages, as the grammars provided in RFCs are often not context free, and such tools provide no support for deferring the parsing of some message fragments. Thus, parsers for network protocol messages have traditionally been implemented by hand. This situation, however, is becoming increasingly impractical, given the variety and complexity of protocols that are continually being developed. For example, the Gaim instant messaging client parses more than 10 different instant messaging protocols.[1] SIP (*Session Initiation Protocol*) [18], which is mainly used in telephony over IP, has a multitude of variants and extensions, implying that SIP parsers must be tolerant of minor variations in the message structure and be extensible. Incorrect or inefficient parsing makes the application vulnerable to denial of service attacks, as illustrated by the "leading slash" vulnerability found in the Flash HTTP Web server [16]. In our experiments (Section 4), we have crashed the widely used SER router [15] for SIP via a stream of 2416 incorrect messages, sent within 17 seconds.

To address the growing complexity of network protocol messages and the inadequacy of standard tools, some parser generators have recently been developed that specifically

[1] GAIM. http://gaim.sourceforge.net

target the kinds of complex data layouts found in network protocol messages. These tools include DATASCRIPT [1] and PacketTypes [11] for binary protocols, and PADS [5], GAPA [3] and binpac [14] for both binary and text-based protocols. However, except GAPA, none of these tools accepts BNF as the input language, and thus, the RFC specification must be translated to another formalism, which is tedious and error prone. Furthermore, such approaches have mainly targeted fixed applications, which do not require fine-grained control over when parsing occurs. While these approaches relieve some of the burden of implementing a network protocol message parser, there still remains a gap between these tools and the needs of applications.

We propose to directly address the issues of correctness and efficiency at the parser generator level. For this, we present a domain-specific language, Zebu, for describing HTTP-like text-based protocol message formats and related processing constraints. Zebu is an extension of ABNF (*Augmented BNF* [4]), the variant of BNF used in RFCs and thus the programmer can simply copy a network protocol message grammar from an RFC to begin developing a parser. Zebu extends ABNF with annotations indicating which message fields should be stored in data structures, and other semantic information. Fields can be declared as `lazy`, which gives control over the time when the parsing of a field occurs. A Zebu specification is processed by a compiler that generates stubs to be used by an application to process network messages. Based on the annotations, the Zebu compiler implements domain-specific optimizations to reduce the memory usage of a Zebu-based application. Besides efficiency, Zebu also addresses robustness, as the compiler performs many consistency checks, and can generate parsing stubs that validate the message structure.

This paper In this paper, we introduce the Zebu language for describing protocol message formats and related processing constraints. Zebu builds on the ABNF notation typically used to describe protocol grammars. We present an assessment of its performance and robustness in the context of SIP, HTTP, and RTSP (*Real Time Streaming Protocol* [2]). Our contributions are as follows:

- We present the motivations for Zebu and the principal decisions that we have taken for its design.
- We have used Zebu to implement parsers for SIP, HTTP, and RTSP. Each parser required less than half a day of development time.
- We have tested Zebu-based SIP and RTSP parsers and standard existing SIP and RTSP parsers on streams of valid and invalid messages based on a message torture suite. While the Zebu-based parsers accept all of the valid messages and reject all of the invalid messages, the existing SIP and RTSP parsers we have tested reject up to about 4% of the valid messages and reject only at most 25% of the invalid ones.
- Finally, we show that the added safety and robustness provided by Zebu does not significantly impact performance. Indeed, our performance evaluation for SIP shows that a Zebu-based parser can be as efficient on average as a hand-crafted one.

The rest of this paper is organized as follows. Section 2 discusses specific characteristics of network protocol message parsing code, illustrating its inherent complexity. Section 3 introduces the Zebu language, and describes the verification of specifications and the generation of parsing stub functions. Section 4 assesses the robustness and performance of Zebu-based parsers. Section 5 described related work and Section 6 concludes.

2. Issues in Developing Network Protocol Parsers

To illustrate the growing complexity of network protocol messages and the inadequacy of existing approaches for creating the associated parsers, we consider the SIP protocol [18]. The SIP message syntax is similar to that of other recent text-based protocols such as HTTP and RTSP. A SIP message begins with a line indicating whether the message is a request (including a protocol method name) or a response (including a return code). A sequence of required and optional headers then follows. Finally, a SIP message can include a body containing the payload. Widely used SIP parsers include that of the SIP Express Router (SER) [15] and the oSIP library[3] used in, *e.g.*, the open PBX Asterisk.[4] Both parsers are hand-written.

We first present an extract of the ABNF specification of the SIP message grammar, and then describe the difficulties in hand-writing the corresponding parser. We next consider to what extent these difficulties are addressed by existing parser generation tools, and describe the issues involved in integrating a parser with a network application.

2.1. ABNF formalism

The ABNF formalism is a modified version of the well known Backus-Naur Form (BNF), and is mainly used for Internet technical specifications. The differences between standard BNF and ABNF involve rule names, repetition, alternatives, order independence, and value ranges. An extract of the ABNF specification of the SIP message grammar is shown in Figure 1. Lines 1 to 6 define the structure of a request line, which appears at the beginning of a message,

[2] http://www.rtsp.org

[3] The GNU oSIP library. http://www.gnu.org/software/osip

[4] Asterisk: The open source PBX. http://www.asterisk.org

and lines 7 to 10 define the structure of the CSeq header field, which is used to identify the collection of messages making up a single transaction.

```
1  Request-Line = Method SP Request-URI SP SIP-Version CRLF
2  Method       = INVITEm / ACKm / OPTIONSm / BYEm
                / CANCELm / REGISTERm / extension-method
3  INVITEm      = %x49.4E.56.49.54.45 ; INVITE in caps
4  Request-URI  = SIP-URI / SIPS-URI / absoluteURI
5  SIP-Version  =  "SIP" "/" 1*DIGIT "." 1*DIGIT
6  extension-method = token
7  CSeq         = "CSeq" HCOLON 1*DIGIT LWS Method
8  LWS          =  [*WSP CRLF] 1*WSP ; linear whitespace
9  SWS          =  [LWS] ; sep whitespace
10 HCOLON       =  *( SP / HTAB ) ":" SWS
```

Figure 1. Extract of the ABNF of the message syntax from the SIP RFC 3261

An ABNF specification consists of a set of derivation rules, each defining a set of alternatives, separated by `/`. An alternative is a sequence of terminals and nonterminals. Among the terminals, a quoted string is case insensitive. Case sensitive strings must be specified as an explicit sequence of character codes, as in the `INVITEm` rule (line 3). ABNF includes a general form of repetition, $n*m$ `X`, that indicates that at least n and at most m occurrences of the terminal or nonterminal `X` must be present. ABNF also defines shorthands such as $n*$ for $n*\infty$, $*n$ for $0*n$, $*$ for $0*\infty$ and n for $n*n$. Therefore, `1*DIGIT` in the `CSeq` rule (line 7) represents a sequence of digits of length at least 1. Brackets are used as a shorthand for `0*1`.

2.2. Hand-writing parsers

The specification of the `CSeq` header in Figure 1 amounts to only four lines of ABNF (lines 7 to 10). However, implementing parsing based on such an ABNF specification efficiently in a general-purpose language often requires many lines of code. For example, SER and oSIP contain about 200 and 340 lines of C and C++ code, respectively, specifically for parsing the `CSeq` header. This `CSeq`-specific code includes operations for reading individual characters from the message, operations for transitioning in a state machine according to the characters that are read, calls to various generic header parsing operations, and error checking code. Among the complexities encountered is the fact that, as shown in Figure 1, a `CSeq` header value can stretch over multiple lines if the continuation line begins with a space or horizontal tab (`WSP`).

In addition to the constraints described by the ABNF specification, the parser developer has to take into account constraints on the message structure that are informally specified in the RFC text. For example, the `CSeq` header includes a *CSeq number*, expressed as any sequence of at least one digit (`1*DIGIT`), and a *CSeq method* (`Method`). The SIP RFC states that the CSeq number must be an unsigned integer that is less than 2^{31} and that the CSeq method must be the same as the method specified in the request line. However, existing hand-written implementations do not always check all these properties. For example, oSIP converts the CSeq number to an integer without performing any verification. If the CSeq number contains any non-numeric characters, the result is meaningless.

2.3. Using parser generators

Existing parser generators for text-based protocols include PADS [5], binpac [14], and the parser generator of the GAPA platform [3]. PADS and binpac use a type-declaration like format for specifying message grammars, while GAPA uses a BNF-like format. Both of these formats require reorganizing the information in the ABNF specification. Binpac is targeted towards a Network Intrusion Detection System, and thus it does not provide high-level support for fine-grained parsing. For example, it provides type abstractions for generic message elements such as digits, but more complex message element decomposition must be performed with regular expressions. The GAPA language, which is not publicly available, is tightly-coupled to a runtime framework specialized for the creation of network analyzers; it is not clear how it could be used to build other kinds of applications. Thus, we take PADS as a concrete example because, of the three, its goals are the most generic. Figure 2 shows a PADS specification corresponding to the four lines of ABNF describing the CSeq header. This specification is in the spirit of the HTTP specification provided by the PADS developers [13].

```
1  bool chkCSeqMethod (request_line_t r, Cseq_t c) {
2     return ( r.method == c.method );
3  }
4  Ptypedef Puint16_FW(:3:) Cseq_number_t :
5     CSeq_t x => { 100 <= x && x < 699 };
6  Pstruct wsp_crlf_t {
7     PString_ME(:"(\\s|\\t)* \\r\\n":) wsp;
8  };
9  POpt wsp_crlf_t o_wsp_crlf_t;
10 Pstruct lws_t {
11    o_wsp_crlf_t wsp_crlf;
12    PString_ME(:"(\\s|\\t)+":) wsp;
13 };
14 POpt lws_t sws_t;
15 Pstruct hcolon_t {
16    PString_ME(:"(\\s|\\t)*":) sp_or_htab;
17    ':';   sws_t   sws;
18 };
19 Pstruct CSeq_t {
20    PString_ME(:"[Cc][Ss][Ee][Qq]":) name;
21    hcolon_t hcolon;   CSeq_number_t number;
22    lws_t lws;         method_t method;
23 };
24 Precord Pstruct SIP_msg {
25    request_line_t request_line;
26    CSeq_t cseq: chkCSeqMethod(request_line,cseq);
27 };
   [...]
```

Figure 2. PADS SIP RFC 3261 specification

A PADS specification describes both the grammar and the data structures that will contain the result of parsing the

message. Thus, the rules of the ABNF specification need to be translated manually into what amount to structure declarations in PADS. As a PADS structure must be declared before it is used, the rule ordering is often forced to be different than that of the ABNF specification. For example, in the ABNF specification, the `CSeq` nonterminal is defined before the `LWS`, `SWS`, and `HCOLON` nonterminals, while in the PADS specification, the structure corresponding to the `CSeq` nonterminal is defined afterwards (line 19). PADS also does not implement the same default parsing strategies as ABNF, and thus *e.g.*, case insensitive strings must be specified explicitly using regular expressions (line 20). Similarly, translating SIP whitespace into PADS requires writing many lines of specifications (lines 6-15), including regular expressions which are known to be time-consuming and difficult to write and error-prone. Finally, the PADS specification must express the various constraints contained in the RFC text. Although PADS allows the developer to define constrained types (lines 5-6), which are used here in the case of the CSeq number (line 21), non-type constraints such as the relationship between the method mentioned in the request line and the method mentioned in the CSeq header must be implemented by arbitrary C code (lines 1-3 and line 26).

Of these issues, probably the most difficult for the programmer is to convert ABNF specifications to regular expressions. Regular expressions for even simple ABNF specifications are often complex and voluminous. For example, a regular expression for a URI has been published that is 45 lines of code.[5] Although a tool has been developed to convert an ABNF specification to a regular expression,[5] in the PADS, GAPA, and binpac specifications that we have seen, the regular expressions appear to have been written by hand, and sometimes do not capture all of the RFC constraints.

2.4. Integrating a parser with an application

The ease of integrating a parser with an application depends on whether the parser parses the fields needed by the application, and whether the result of this parsing is stored in appropriate data structures. We consider the issues that arise when using the handwritten oSIP and SER parsers, and when using a parser generated by a tool such as PADS.

oSIP parses the fixed set of required SIP header fields, and separates the rest of the message into pairs of a header field name and the corresponding raw unparsed data. Applications that do not use all of the required header fields still incur the time cost of parsing them (see Section 4). Applications that use the many SIP extensions must parse these header fields themselves. The former increases the application time and space requirements, which can be critical in the case of in-network applications such as proxies, while the latter leaves the application developer to develop complex parsing code on his own.

[5] Abnf2Regex. http://cvs.m17n.org/∼akr/abnf/.

SER provides more fine-grained parsing than oSIP, as it parses only those header fields that are requested by the application. By default, however, SER only gives direct access to the top-level subfields of a header, such as the complete URI. To extract, *e.g.*, only the host portion of a URI, the programmer must intervene. One approach is for an application using SER to reparse the subfield, to obtain the desired information. SER applications are written using a domain-specific language targeted towards routing, which does not provide string-matching facilities. Nevertheless, SER provides an escape from this language, allowing a SER application to invoke an arbitrary shell script. A SER application can thus invoke a script written in a language such as Perl to extract the desired information. This approach, however, incurs a high performance penalty for forking a new process (see Section 4) and compromises the safety benefits of using the SER language. Another approach is to use the SER extension framework, which, *à la* Apache,[6] allows integrating new modules into the parsing process. Although efficient, this approach requires the programmer to write low level C code that conforms to rather contorted requirements. Again, incorrect behavior inside a module may compromise the robustness of the whole application.

Finally, parser generators such as PADS allow the developer to construct the parser such that it parses only as much of the message as is needed. However, the generated data structures directly follow the specified parsing rules, implying that accessing message fields often requires long chains of structure field references. Furthermore, all of the parsed data is stored, which increases the memory footprint.

3. Robust Parser Development with Zebu

We now present the Zebu language for describing HTTP-like text-based protocol message formats and related processing constraints. Zebu is based on ABNF, as found in RFCs, and extends it with annotations indicating which message fields should be stored in data structures and other semantic attributes. These annotations express both constraints derived from the protocol RFC and constraints that are specific to the target application. From a Zebu specification, a compiler automatically generates stubs to be used by the application to process network messages.

The features of Zebu are driven by the kinds of information that an application may want to extract from a network protocol message. We first consider the features that are needed to do this processing robustly and efficiently, and then present the corresponding annotations that the programmer must add to the ABNF specification so that the Zebu compiler can generate the appropriate stub functions. Finally, we describe the Zebu compiler, which performs both verification and code generation, and the process of constructing an application with Zebu.

[6] Apache. HTTP server project. http://www.apache.org.

3.1. Issues

A HTTP-like text-based network message consists of a command line, a collection of header fields, and a message body. The command line indicates whether the message is a request or a response, and identifies basic information such as the version of the protocol and the method of a request message. A header field specifies a protocol-specific key and an associated value, which may be composed of a number of subfields. Finally, the message body consists of free text whose structure is typically not specified by the protocol. Decomposing it further falls out of the scope of Zebu.

From the contents of a message, an application may need to determine whether the message is a request or a response, to detect the presence of a particular header field, or to extract command-line or header-field subfields. Each of these operations involves retrieving a command line or header field, and potentially accessing its contents. In a HTTP-like text-based protocol, each command line or header field normally occupies one or more complete lines, where each line after the first begins with a special continuation character. Thus, as exemplified by the very efficient SIP parser SER, a parser can be constructed in two levels: a top-level parser that simply scans each line of the message until it reaches the desired command line or header field, and a collection of dedicated parsers that process each type of command line or header field. The dedicated parsers must respect both the ABNF specification and any constraints specified informally in the RFC. To avoid reparsing already parsed message elements for each requested parsing operation, the parser should save all parsed data in data structures for later use, ideally in the format desired by the application.

This analysis suggests that to enable the Zebu compiler to generate a useful and efficient parser, the programmer must annotate the ABNF specification obtained from an RFC with the following information: (1) An indication of the nonterminal representing the entry point for parsing each possible command line and header field. (2) A specification of any constraints on the message structure that are informally described by the RFC. (3) An indication of the message subfields that will be used by the application. The first two kinds of annotations are generic to the protocol, and can thus be reused in generating parsers for multiple applications. The third kind of annotation is application-specific. This kind of annotation can be viewed as a simplified form of the action that can be specified when using `yacc` and other similar parser generators, in that it allows the programmer to customize the memory layout used by the parser to the specific needs of the application.

3.2. Annotating an ABNF specification

We present the three kinds of annotations required by Zebu, using as an example an extract of the Zebu specification of a SIP parser, as shown in Figure 3.

```
1  message sip3261 {
2   request {; Request only
3    requestLine = Method:method SP  Request-URI:uri
4                  SP SIP-Version
5    header CSeq { CSeq.method == requestLine.method }
6    header Max-Forwards { mandatory }
7   }
8   response {; Response only
9    statusLine = SIP-Version SP Status-Code:code
10                SP Reason-Phrase:rphrase
11  }
12  enum Method = INVITEm / ACKm / OPTIONSm / BYEm
13              / CANCELm / REGISTERm / extension-method
14  extension-method = token
15  INVITEm = %x49.4E.56.49.54.45 ; INVITE in caps
16  struct Request-URI = SIP-URI / SIPS-URI / absoluteURI
17        {lazy}
18  uint16 Status-Code = Informational / Redirection
19                     / Success / Client-Error
20                     / Server-Error / Global-Failure
21                     / extension-code
22  uint16 Global-Failure = "600"/ "603"/ "604"/ "606"
23  uint16 extension-code = 3DIGIT
24         {extension-code>=100 && extension-code<=699}
25  header CSeq = 1*DIGIT:number as uint32 LWS Method:method
26  header Max-Forwards = 1*DIGIT:value as uint32 {mandatory}
27  header To { "to" / "t" } =
28     ( name-addr / addr-spec:uri ) *( SEMI to-param )
29     { mandatory }
30  name-addr = [display-name] LAQUOT addr-spec:uri RAQUOT
31  struct addr-spec = SIP-URI / SIPS-URI / absoluteURI
32        { lazy }
33  [...]
```

Figure 3. Excerpt of the Zebu specification for the SIP protocol

Parser entry points The Zebu programmer annotates the rule for parsing the command line of a request message with `requestLine`, the rule for parsing the command line of a response message with `statusLine`, and the rules for parsing each kind of header with `header`. Because a command line or header field cannot contain another command line or header field, the nonterminals for these lines are no longer useful. In the case of a command line, the nonterminal is simply dropped. Thus, for example, the ABNF rule for the `Request-Line` nonterminal (Figure 1, line 1) is transformed into the following Zebu rule (Figure 3, line 3):

```
requestLine = Method SP Request-URI SP SIP-Version
```

In the case of a header field, the description of the key is moved from the right-hand side of the rule to the left, where it replaces the nonterminal, resulting in a rule whose structure is suggestive of a key-value pair. For example, the ABNF `CSeq` rule on line 7 of Figure 1 is reorganized into the following Zebu rule (Figure 3, line 25)

```
header CSeq = 1*DIGIT LWS Method
```

(The delimiter `HCOLON` is also dropped, as it is a constant of the protocol). Some header fields, such as the SIP `To` header field, can be represented by any of a set of keys. In this case, the header is given a name, which is followed by the ABNF specification of the possible variants, in braces, as shown in

line 27 of Figure 3. As in ABNF, the matching of the header key, and any other string specified by a Zebu grammar, is case insensitive.

RFC constraints The text of the RFC for a protocol typically indicates how often certain header fields may appear, whether header fields can be modified, and various constraints on the values of the header subfields. The Zebu programmer must annotate the corresponding ABNF rules with these constraints. Constraints are specified in braces at the end of a grammar rule. Possible atomic constraints are that a header field is mandatory (`mandatory`) and that a header field can appear more than once (`multiple`). For example, in the SIP specification, the header `To` is specified to be mandatory (line 27). More complex constraints can be expressed using C-like boolean expressions. For example, in Section 2.2, we noted that in a request message, the method mentioned in the command line must be the same as the method mentioned in the CSeq header. This constraint is described in line 5.

Some constraints on header fields are specific to either request or response messages. Accordingly, the Zebu programmer must group the request line and its associated constraints in a *request block*, and the status line and its associated constraints in a *response block*. In the case of SIP, the request block (lines 2-7) indicates for example that the `Max-Forwards` header is mandatory (line 6). The constraints in the response block (lines 9-11) have been elided.

Subfields used by the application The parsing functions generated by the Zebu compiler create a data structure for each command line or header field that is parsed. By default, this data structure contains only the type of the command line or header field and a pointer to its starting point in the message text. When the application will use a certain subfield of the command line or message header, the Zebu programmer can annotate the nonterminal deriving this subfield with an identifier name. This annotation causes the Zebu compiler to create a corresponding entry in the enclosing command line or header field data structure. For example, in line 3, the Zebu programmer has indicated that the application needs to use the method in the command line, which is given the name method, and the URI, which is given the name uri.

By default, a subfield is just represented as a pointer to the start of its value and its size in the message text. Often, however, the application will need to use the value in some other form, such as an integer. The Zebu programmer can additionally specify a type for a named value, either at the nonterminal reference or at its definition. For example, in line 25 the CSeq number is specified as being a uint32. Nonterminals can also be specified as structures (`struct`), unions (`union`), and enumerations (`enum`).

An application may use the information in certain subfields only in some exceptional cases. The Zebu annotation `lazy` specifies that a specific subfield should not be parsed until requested by the application. In the SIP specification, `Request-URI` has this annotation (line 16).

3.3. The Zebu compiler

The Zebu compiler verifies the consistency of the ABNF specification and the annotations added by the programmer, and then generates stub functions allowing an application to parse the command line and header fields and access information about the parsed data. The Zebu compiler is around 3700 lines of OCaml code. A run-time environment defining various utility functions is also provided, and amounts to around 700 lines of C code.

Verifications Although RFCs are widely published and form the *de facto* standard for many protocols, we have found some errors in RFC ABNF specifications. These are simple errors, such as typographical errors, but still they complicate the process of translating an ABNF specification into code . The Zebu compiler checks basic consistency properties of the ABNF specification: that there is no omission (*i.e.*, each referenced rule is defined), that there is no double definition, and that there are no cycles.

Additionally, the annotations provided by the Zebu programmer must be consistent with the ABNF specification. For example, in line 22, the nonterminal `Global-Failure` is annotated with `uint16`. This non-terminal is specified to be an alternation of strings, and thus the Zebu compiler checks that each element of this alternation represents an unsigned integer that is less than 2^{16}.

Code generation An application does not use the data structures declared in a Zebu specification directly, but instead uses stub functions generated by the Zebu compiler. The Zebu compiler can be configured to create stub functions that perform either full validation or minimal validation of the message structure. The use of stub functions allows parsing to be carried out lazily, so that only as much data is parsed as is needed to fulfill the request of a given stub function call. As illustrated in Figure 4a, stub functions are generated for determining the type of a message (request or response), for parsing the command line and the various headers, for accessing individual header subfields, and for managing the parsing of subfields designated as `lazy`.

The parsing functions generated by the Zebu compiler use the two-level parsing strategy described in Section 3.1. Header-specific parsers use the PCRE[7] library for matching the regular expression of a header value that has been derived from the ABNF specification. The parsing functions

[7] PCRE - perl custom regular expressions. http://www.pcre.org/

```
sip3261 sip3261_init();
void    sip3261_parse(sip3261, char *, int);
void    sip3261_parse_headers(sip3261, E_Headers);
void    sip3261_parse_addr_spec(T_Lazy_addr_spec);
T_bool  sip3261_isRequest(sip3261);
T_bool  sip3261_isResponse(sip3261);
T_Str   sip3261_Option_Str_getVal(T_Option_Str);
T_RequestLine sip3261_get_RequestLine(sip3261);
T_RequestLine  sip3261_get_RequestLine(sip3261);
T_header_From  sip3261_get_header_From(sip3261);
T_Method       sip3261_RequestLine_getMethod(T_RequestLine);
T_MethodEnum   sip3261_Method_getType(T_Method);
T_Str          sip3261_Method_getValue(T_Method);
T_Lazy_addr_spec sip3261_header_From_getUri(T_header_From);
T_addr_spec    sip3261_Lazy_Addr_spec_getParsed(T_Lazy_addr_spec);
T_Option_Str   sip3261_Addr_spec_gethost(T_addr_spec);
```

4a. Generated stubs

```
1   sip3261 msg = sip3261_init();
2   sip3261 msg = sip3261_parse(msg, buf, len);
3   if (sip3261_isRequest(msg)) {  // Process only request messages
4     T_RequestLine requestLine = sip3261_get_RequestLine(msg);
5     if (sip3261_Method_getType(sip3261_RequestLine_getMethod(requestLine)) == E_INVITEm) { // Filter INVITE methods
6       sip3261_parse_headers(msg, E_HEADER_FROM); // We parse only the header From
7      T_Lazy_addr_spec l_addr_spec = sip3261_header_From_getUri(sip3261_get_header_From(msg));
8      sip3261_parse_addr_spec(l_addr_spec);
9      T_Option_Str host = sip3261_Lazy_Addr_spec_getParsed(l_addr_spec);
10      if (sip3261_Option_Str_isDefined(host)) {  // host may be undefined in some cases, check it and log its value
11         mylog(sip3261_Option_Str_getVal(host));
12  }}}
```

4b. Application logic

Figure 4. Fragment of a Zebu-based SIP message statistics reporting application

contain run-time assertions that check the constraints specified in the RFC. Once a header is parsed and checked, its named subfields, if any, are converted to the specified types and stored in the data structure associated with the header. The values of the named subfields can then be accessed using the "get" stub functions.

3.4. Developing an application with Zebu

The developer defines the application logic as an ordinary C program, using the stub functions to access information about the message content. Figure 4b illustrates the implementation of an application that extracts the host information from the URI stored in the `From` header field of an `INVITE` message. This kind of operation is useful in, *e.g.*, an intrusion detection system, which searches for certain patterns of information in network messages.

The application uses the stubs generated from the SIP message grammar specification to access the required information. The application initially uses the functions `sip3261_Method_getType` and `sip3261_RequestLine_getMethod` to determine whether the current message is an `INVITE` request (line 5). If so, it uses the function `sip3261_parse_headers` to parse the `From` header field (line 6), and then the functions `sip3261_header_From_getUri` and `sip3261_get_header_From` to extract the URI (line 7). Line 32 of the Zebu SIP specification indicates that the parsing of the URI should be lazy, so the function `sip3261_Lazy_Addr_spec_getParsed` is used to force the parsing of this subfield (line 9). After a check that the host name is present (line 10), its value is extracted using the function `sip3261_Option_Str_getVal` in line 11.

Overall, due to the annotations in the Zebu specification, stub functions are available to access exactly the message fragments needed by the application. Similarly, memory usage is limited to the application's declared needs.

4. Experiments

A robust network application must accept valid messages, to provide continuous service, and reject invalid messages, to avoid corrupting its internal state, while being efficient. As the parser is the front-line in the treatment of network messages, it plays a key role in providing this robustness and efficiency. We evaluate robustness by comparing the behavior of a Zebu-based parser and a variety of existing parsers on valid and invalid SIP and RTSP messages. We evaluate efficiency by measuring the parsing time of various SIP parsers when applied to messages extracted from a real SIP trace. We also measure the parsing time for isolated HTTP messages.

For SIP, we compare the Zebu-based parsers with the oSIP and SER parsers described in Section 2. For RTSP, we compare the Zebu-based parsers with the parser in the widely used VLC streaming server,[8] and the parser provided by the LiveMedia library.[9] Figure 1 shows the sizes of these existing parsers and of the ABNF and Zebu specifications of the SIP and RTSP message grammars. The Zebu specifications are up to 12 times smaller than the other parsers. The Zebu specifications are, however, about 50% longer than the corresponding ABNF specifications, because they include rules that are mentioned only by reference to another RFC in the ABNF.

4.1. Robustness evaluation

To compare the robustness of Zebu-based SIP and RTSP applications to applications based on hand-crafted parsers,

[8] The VideoLan project. http://www.videolan.org

[9] LiveMedia. http://www.livemediacast.net/about/library.cfm

Protocol	ABNF size	Zebu spec. size	Parser	Parser size
SIP	≈ 700	1081	oSIP	11982
			SER	13277
RTSP	≈ 200	330	VLC	≈ 1200
			LiveMedia	≈ 1000

Table 1. Sizes (LOC) of the SIP and RTSP message grammars, and of existing parsers.

we create minimal logging applications that parse the fields designated as mandatory by the SIP and RTSP protocols. The Zebu-based applications, `log-Zebu-SIP` and `log-Zebu-RTSP`, for SIP and RTSP respectively, consist of a few lines of C code that use the stubs generated by the Zebu compiler to access network messages, analogous to the code in Figure 4b. We configure the Zebu compiler to generate stub functions performing full validation. The SER application, `log-SER`, is written using the SER configuration language to access the information in the various fields. The other applications, `log-oSIP` using oSIP, `log-VLC` using VLC, and `log-LiveMedia` using LiveMedia, are written in C using the appropriate API functions provided by the given parser.

In our experiments, we test the applications on streams of valid or invalid messages. For the SIP applications, these messages are created according to the guidelines provided by the SIP Torture Test Message RFC [19] and for the RTSP applications, we create messages in the same spirit. Valid but potentially problematic messages suggested by the SIP Torture Test Message RFC are typically those with complex URIs or with special characters such as a space or semicolon protected with a backslash. Invalid messages suggested by the SIP Torture Test Message RFC somehow violate the structural or value constraints of the grammar. An example is a SIP response status code of 2 or 4 digits, whereas a valid response status code always consists of 3 digits.

Valid messages Figure 2 shows that up to about 4% of the valid messages that we have constructed based on the SIP Torture Test Message RFC are rejected by hand-crafted SIP parsers. We have also observed that oSIP crashes on some valid `INVITE` messages. The Zebu-based SIP parser strictly follows the message grammar.

		Valid messages	Rejected messages	% rejected messages
SIP	`log-Zebu-SIP`	549	0	**0.0**%
	`log-oSIP`		21	**3.9**%
	`log-SER`		2	**0.4**%

Table 2. Coverage for valid SIP messages

We have made an analogous experiment with the RTSP applications, but the VLC and LiveMedia parsers are quite lax in their parsing of the message elements, such as the URI, that are covered by the SIP Torture Test RFC, and thus all three applications accept all of the messages.

Invalid messages Figure 3 shows that the Zebu-based applications detect every invalid message, but none of the hand-crafted parsers detects more than about 25% of the invalid messages. A parser that does not detect invalid messages may return unexpected data to the application, causing it to crash or malfunction. For example, in the case of SIP we have crashed SER via a stream of 2416 incorrect messages, sent within of 17 seconds. Because SER is widely used for telephony, which is a critical service, the ability to crash the server is unacceptable.

		Invalid messages	Detected messages	% detected messages
SIP	`log-Zebu-SIP`	5976	5976	**100.0**%
	`log-oSIP`		1020	**17.1**%
	`log-SER`		1512	**25.3**%
RTSP	`log-Zebu-RTSP`	2730	2730	**100.0**%
	`log-VLC`		4	**0.1**%
	`log-LiveMedia`		748	**27.4**%

Table 3. Coverage for invalid SIP and RTSP messages

4.2. Performance Evaluation

We now compare the performance of Zebu-based parsers to that of hand-crafted ones. We first present micro-benchmarks that we performed for HTTP, and then present results for SIP based on real message traces.

Our experiments used a Pentium 4 (2.66GHz) as a server, and an Athlon Mobile 1.6GHz as a client. The client continuously extracts messages from a trace of generated or real messages and sends them to the server. As we are only considering the parsing time, the messages frequency is irrelevant.

HTTP micro-benchmarks The PADS distribution includes a specification of a parser for HTTP/1.1. This specification consists of a combination of PADS structure type declarations and regular expressions, and amounts to 2000 lines of PADS code. Because of some constraints in the PADS language, this specification is significantly different from the ABNF in the HTTP RFC. The Zebu HTTP specification, in contrast is only 500 lines, and is very close to the ABNF. The PADS compiler generates about 150000 lines of C code from the PADS HTTP specification, whereas the Zebu compiler generates only about 2400 from the Zebu specification.

To measure the HTTP message treatment time with these Zebu- and PADS-generated parsers and with standard parsers, we have developed applications that log the domain name of the `HOST` header field for use with the Zebu-based parser, the PADS-based parser, and Apache. The applications using the Zebu-based parser and the PADS-based parser use the TCP server provided with SER. The Zebu-based application is 27 lines of C code, the PADS-based

application is 50 lines of C code, and the Apache-based application is 9 lines of Apache configuration language code. For the Zebu-based application, parsing per message with full validation requires 170,000 cycles on average. For the PADS-based application, parsing is 1500 times slower, at about 260,000,000 cycles per message on average. For Apache, which has a very highly hand-optimized parser, parsing requires 24,000 cycles per message on average, representing 14% of the time needed by the Zebu-based parser.

SIP To experiment with real SIP traces, we have implemented four versions of the SIP message statistics reporting application described in Section 3.4. Each application records the host information of the URI stored in the `From` header field of an `INVITE` message. This message statistics reporting application illustrates the case where an application such as a intrusion detection system needs to access a fragment of a header subfield.

The first version (`SER-module`) is implemented as a dedicated SER module to obtain full access to the internal data structures of SER. This module amounts to 150 lines of C code and requires 22 lines to be added in the configuration file of the server. The second version (`SER-exec`) is written using the SER configuration language and relies on the escape mechanism provided by SER to invoke `sed` to extract the host information. This version consists of 12 lines of SER configuration language code. The third version (`oSIP`) is implemented on top of the oSIP SIP stack. It consists of 40 lines of C code that configure the stack and call the different functions the stack provides. The fourth version (`Zebu-full`) is the Zebu-based application shown in Figure 4b with full validation enabled. The last version (`Zebu-minimal`) is similar to the previous one, except that only fields that are annotated to be used by the application are validated. In both cases, 28 lines of C code are required to call the stubs generated by the Zebu compiler.

To explore the time required for parsing various kinds of messages, we have collected a real trace of SIP messages at the University of Bordeaux. This trace represents one day of IP telephony traffic amounting to around 3000 messages. Figure 4 compares the parsing time for each of the applications to that of `Zebu-minimal`.

SER uses the efficient two-level parsing strategy described in Section 3.1, to parse only the header fields that are relevant to the application. The parsing done by `SER-module` is particularly efficient, up to three times faster than `Zebu-minimal` in the case of request messages, as the information required by the application is already available in the SER internal data structures. However, it is 50% slower than the parsing done by `Zebu` in the case of response messages. SER is directed towards routing applications, and thus it always parses the `Via` header, which is essential in the routing process, although irrelevant to our application. Thus, Zebu can provide better performance by being more closely tailored to the needs of the application, and retains safety, which is lost when using SER modules, as described in Section 2.4.

The parsing done by `SER-exec` is roughly as efficient as that done by `SER-module` for the non-`INVITE` message. The parsing done by `SER-exec` for the `INVITE` messages, on the other hand, is about 217 times slower because it forks a `sed` process. Despite the bad performance in this case, the use of the configuration language of SER remains a valid approach, because it provides ease of programming and safety.

The parsing done by `oSIP` is between 5 and 10 times slower than the parsing done by `Zebu-minimal` for request messages and over 21 times slower for response message. In both cases, oSIP parses the six required SIP headers (plus two more required headers in the case of a `REGISTER` message) and stores pointers to the starting point of each sub-field. As the application requests information about the `INVITE` header field, oSIP additionally copies its subfields into a data structure that is provided to the application, roughly doubling the execution time.

5. Related Work

Parser generators such as DATASCRIPT [1], PacketTypes [11], PADS [5], GAPA [3] and binpac [14] have been developed to address the growing complexity of network protocol messages. However, as described in Section 2, these tools do not meet all the requirements of network application developers. APG[10] is a parser generator that accepts ABNF directly. Semantic actions are specified via callback functions rather than annotations on the grammar. We have found such callback functions to be somewhat heavyweight, in our experience with APG. Furthermore, APG is not specific to HTTP-like text-based protocols, and thus cannot implement the two-level parsing strategy outlined in Section 3.1. We have found this strategy essential to obtaining good performance.

Domain-specific languages have been used successfully in various application domains including operating systems [10, 12] and networks [6, 7]. Several of these languages have explicitly targeted improving system robustness. Devil, in the domain of device-driver development, provides high-level abstractions for specifying the code for interacting with the device, and performs a number of compile-time and (optional) run-time verifications to check that the specifications are consistent [17]. Promela++ [2] for specifying network protocols, can be translated automatically both into the model checking language Promela [8] and into efficient C code, thus easing the development of a protocol implementation that is both verified and efficient.

[10] APG. http://www.coastocoastresearch.com/

		SER-module		SER-exec		oSIP		Zebu-full		Zebu-minimal	
		Cycles	**Ratio**	**Cycles**	**Ratio**	**Cycles**	**Ratio**	**Cycles**	**Ratio**	**Cycles**	**Ratio**
Request	ACK	25 460	32%	22 690	29%	377 298	481%	81 796	104%	78 406	100%
	BYE	22 540	42%	36 608	69%	522 156	984%	77 468	146%	53 060	100%
	CANCEL	22 732	46%	52 996	107%	336 036	676%	84 512	170%	49 680	100%
	INVITE	34 868	43%	7 593 550	9 438%	525 654	653%	81 670	102%	80 456	100%
	OPTIONS	25 872	36%	24 176	34%	339 016	478%	76 144	107%	70 908	100%
	REGISTER	31 016	50%	31 096	50%	465 978	751%	65 672	106%	62 028	100%
Response		35 608	151%	35 716	152%	505 312	2 150%	31 806	135%	23 508	100%

Table 4. Performance of the SIP message statistics reporting applications (time in cycles, ratio as compared to `Zebu-minimal`)

6. Conclusion

In this paper, we have presented the Zebu declarative language for describing protocol message formats and related processing constraints. Zebu builds on the ABNF notations typically used in RFCs to describe protocol grammars. In evaluating Zebu, we have focused on analyzing the improvement in robustness that it provides. For this, we have defined a set of rules based on a message torture suite to generate valid and invalid SIP and RTSP messages and applied existing parsers and Zebu-generated parsers to them.

Our experiments show that nearly 4 times more erroneous messages are detected by the Zebu-based parser than by widely-used hand-written parsers. In the case of SIP, we were able to crash the widely used SER parser [15] via a stream of 2416 incorrect messages, sent within a space of 17 seconds. Because SER is used for telephony the ability to crash the server is unacceptable. We have also found valid messages that are not accepted by the SER and oSIP parsers, which can similarly have a critical impact. Finally, we have shown that among the approaches providing safety and robustness guarantees, Zebu provides good performance.

Availability: The Zebu compiler and experiments mentioned in the paper are available at the following web page: `http://phoenix.labri.fr/software/zebu`

References

[1] G. Back. DataScript - a specification and scripting language for binary data. In *ACM Conference on Generative Programming and Component Engineering*, pages 66–77, USA, 2002.

[2] A. Basu, J. G. Morrisett, and T. von Eicken. Promela++: A language for constructing correct and efficient protocols. In *Proceedings IEEE Conference on Computer Communications*, pages 455–462, San Francisco, CA, USA, Mar. 1998.

[3] N. Borisov, D. J. Brumley, H. J. Wang, J. Dunagan, P. Joshi, and C. Guo. A generic application-level protocol analyzer and its language. In *14th Annual Network & Distributed System Security Symposium*, USA, Feb. 2007.

[4] D. Crocker and P. Overell. Augmented BNF for Syntax Specifications: ABNF. IETF: RFC 2234, Nov. 1997.

[5] K. Fisher and R. Gruber. PADS: a domain-specific language for processing ad hoc data. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 295–304, Chicago, IL, USA, June 2005. ACM.

[6] D. Gay, P. Levis, J. R. von Behren, M. Welsh, E. A. Brewer, and D. E. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 1–11, San Diego, California, June 2003.

[7] M. Hicks, P. Kakkar, J. Moore, C. Gunter, and S. Nettles. PLAN: A packet language for active networks. In *Proceedings of the ACM International Conference on Functional Programming Languages*, pages 86–93. ACM, June 1998.

[8] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[9] S. C. Johnson. Yacc: Yet another compiler compiler. Technical report, Bell Telephone Laboratories, 1975.

[10] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 75–90, Brighton, UK, Oct. 2005.

[11] P. J. McCann and S. Chandra. Packet types: Abstract specifications of network protocol messages. In *ACM SIGCOMM 2000 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 321–333, Stockholm, Sweden, Aug. 2000.

[12] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for Hardware Programming. In *4th Symposium on Operating Systems Design and Implementation*, pages 17–30, USA, Oct. 2000.

[13] PADS distribution. `http://www.padsproj.org/download-src-1.03.html`.

[14] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: a yacc for writing application protocol parsers. In *Proceedings of the 6th ACM SIGCOMM on Internet measurement*, pages 289–300, Rio de Janeiro, Brazil, Oct. 2006.

[15] A. Pelinescu-Onciul, J. Janak, and J. Kuthan. SIP express router (SER). *IEEE Network Magazine*, 17(4):9, 2003.

[16] X. Qie, R. Pang, and L. L. Peterson. Defensive programming: Using an annotation toolkit to build DoS-resistant software. In *5th Symposium on Operating System Design and Implementation*, pages 45–60, USA, Dec. 2002.

[17] L. Réveillère and G. Muller. Improving driver robustness: an evaluation of the Devil approach. In *The International Conference on Dependable Systems and Networks*, pages 131–140, Sweden, July 2001. IEEE Computer Society.

[18] Rosenberg, J. et al. SIP: Session Initiation Protocol. RFC 3261, IETF, June 2002.

[19] R. Sparks, A. Hawrylyshen, A. Johnston, J. Rosenberg, and H. Schulzrinne. Session initiation protocol (SIP) torture test messages. IETF: RFC 4475, May 2006.