HAL
open science

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

Laurent Burgy, Laurent Réveillère, Julia L. Lawall, Gilles Muller

To cite this version:

Laurent Burgy, Laurent Réveillère, Julia L. Lawall, Gilles Muller. A Language-Based Approach for Improving the Robustness of Network Application Protocol Implementations. [Research Report] 2007. inria-00140964v1

HAL Id: inria-00140964

https://inria.hal.science/inria-00140964v1

Submitted on 11 Apr 2007 (v1), last revised 24 Apr 2007 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

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

Laurent Burgy — Laurent Réveillère — Julia Lawall — Gilles Muller

N° ????

Février 2007

Thème COM

Rapport de recherche

inria-00140964, version 1 - 11 Apr 2007

ISSN 0249-6399 ISRN INRIA/RR--????--FR+ENG

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

Laurent Burgy , Laurent Réveillère , Julia Lawall , Gilles Muller

Thème COM — Systèmes communicants
Projets Phoenix et Obasco

Rapport de recherche n° ???? — Février 2007 — 16 pages

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 describing protocol message formats and related processing constraints. From a Zebu specification, a compiler automatically generates stubs to be used by an application to parse network messages. Zebu is easy to use, as it builds on notations used in RFCs to describe protocol grammars. Zebu is also efficient, as the memory usage is tailored to application needs and message fragments can be specified to be processed on demand. Finally, Zebu-based applications are robust, as the Zebu compiler automatically checks specification consistency and generates parsing stubs that include validation of the message structure. Using a mutation analysis in the context of SIP and RTSP, we show that Zebu significantly improves application robustness.

Key-words: Langage métier, protocoles réseau, analyze de mutation

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 ORSAY Cedex (France)
Téléphone : +33 1 72 92 59 00 — Télécopie : +33 1 60 19 66 08

Une approche langage pour améliorer la robustesse de l'implémentation de protocoles réseaux applicatifs

Résumé : Pour être sûr et robuste, le fonctionnement d'un réseau doit reposer sur des implémentations d'applications sans faille. Les protocoles réseau étant de plus en plus complexes, écrire manuellement le code qui prend en charge leurs messages devient de plus en plus difficile et sujet à erreurs.

Dans ce papier, nous présent un langage métier, Zebu, pour décrire le format des messages d'un protocole réseau et les contraintes de traitement associées. D'une spécification Zebu, un compilateur génère automatiquement des talons à utiliser par une application pour l'analyze grammaticale de messages réseau. Zebu is simple d'usage, utilisant les mêmes notations que celles utilisées dans les RFCs pour décrire les grammaires de protocoles. Zebu est efficace, l'implantation mémoire étant calquée sur les besoins de l'application et les fragments du message pouvant être traités à la demande. Enfin, les applications basées sur Zebu sont robustes, le compilateur vérifiant la consistence de la spécification et les talons générés étant incluant la validation de la structure du message. En utilisant une analyze de mutation dans le contexte de SIP et RTSP, nous montrons que Zebu améliore de manière significative la robustesse des applications.

Mots-clés : DSL, network protocols, mutation analysis

Contents

1 Introduction **3**

2 Issues in developing network protocolparsers **4**
2.1 ABNF formalism . . . 5
2.2 Hand-writing parsers . . . 5
2.3 Using parser generators . . . 5
2.4 Integrating a parser with anapplication . . . 6

3 Robust Parser Development with Zebu **7**
3.1 Issues . . . 7
3.2 Annotating an ABNF specification . . . 8
3.3 The Zebu compiler . . . 9
3.4 Developing an application with Zebu . . . 11

4 Experiments **11**
4.1 Robustness evaluation . . . 11
4.2 Performance Evaluation . . . 13

5 Related Work **14**

6 Conclusion **14**

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 [24]. 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. The syntax of network protocol messages is typically specified in a RFC (*Request for Comments*) using a variant of BNF known as ABNF (*Augmented BNF*) [6]. 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. Furthermore, some kinds of message processing may not use all fragments 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 [23]. 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` [15]. 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 [10]. The message grammar in the IMAP RFC is about 500 lines of ABNF, and includes external references to others RFCs. SIP (*Session Initiation Protocol*) [26], 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 [24]. In our experiments (Section 4), we have crashed the widely used SER parser [23] 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 target the kinds of complex data layouts found in network protocol messages. These tools include DATASCRIPT [3] and PacketTypes [18] for binary protocols,

and PADS [9], GAPA [5] and binpac [22] for both binary and text-based protocols. However, none of these approaches accepts ABNF 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 application protocol analyzers, which parse a fixed portion of the message and then proceed to some analysis phase. Thus, they do not provide fine-grained control over the time 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. To this end, 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, implying that 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, such as the type of the value expressed by a field, constraints on the range of its value, and whether certain fields are mandatory or optional. Fields can additionally be declared as `lazy`, which gives control over the time when the parsing of a field occurs. A Zebu specification is then 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 generates parsing stubs that validate the message structure.

This paper In this paper, we present the Zebu language and an assessment of its performance and robustness in the context of the SIP and RTSP (Real Time Streaming Protocol [27]) network protocols. Our contributions are as follows:

- We introduce a declarative language, named Zebu, for describing protocol message formats and related processing constraints. Zebu builds on the ABNF notation typically used in RFCs to describe protocol grammars.
- We have defined a test methodology based on a mutation analysis for evaluating the robustness improvement induced by Zebu.
- We have applied our test methodology existing and Zebu-based SIP and RTSP parsers. While the Zebu-based parsers reject 100% of the invalid mutated messages, none of the existing parsers that we have tested detects more than about 25% of the injected mutants.
- Finally, we show that the added safety and robustness provided by Zebu does not significantly impact performance. Indeed, our performance evaluation 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 to creating the associated parsers, we consider the SIP protocol [26]. 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) [23] and the oSIP library [20] used *e.g.*, in the open PBX Asterisk [29]. Both parsers are hand-written.

We first present an extract of the ABNF specification of the SIP message grammar, and then describe the difficulty of 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.

```
Request-Line = Method SP Request-URI SP SIP-Version CRLF                    1
Method       = INVITEm / ACKm / OPTIONSm / BYEm                             2
               / CANCELm / REGISTERm                                        3
               / extension-method                                           4
INVITEm      = %x49.4E.56.49.54.45 ; INVITE in caps                         5
Request-URI  = SIP-URI / SIPS-URI / absoluteURI                             6
SIP-Version    =  "SIP" "/" 1*DIGIT "." 1*DIGIT                             7
extension-method = token                                                    8
[...]
CSeq = "CSeq" HCOLON 1*DIGIT LWS Method                                    10
LWS  =  [*WSP CRLF] 1*WSP ; linear whitespace                              11
SWS  =  [LWS] ; sep whitespace                                             12
HCOLON  =  *( SP / HTAB ) ":" SWS                                          13
```

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

2.1 ABNF formalism

An extract of the ABNF specification of the SIP message grammar is shown in Figure 1. Lines 1 to 8 define the structure of a request line, which appears at the beginning of a message, and lines 10 to 13 define the structure of the CSeq header field, which is used to identify the collection of messages making up a single transaction.

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 5). 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 10) 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. However, implementing parsing based on such an ABNF specification efficiently in a general-purpose language such as C or C++ 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 text of the RFC. 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 requirements. 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 a meaningless value.

2.3 Using parser generators

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. We take PADS as a concrete example. 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 [21].

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 are translated 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 26). PADS also does not implement

```
bool chkCseqMethod (request_line_t r, Cseq_t c) {                                  1
   return ( r.method == c.method );                                                2
}                                                                                  3

Ptypedef Puint16_FW(:3:) Cseq_number_t :                                           5
   CSeq_t x => { 100 <= x && x < 699 };                                            6

Pstruct wsp_crlf_t {                                                               8
   PString_ME(:"(\\s|\\t)* \\r\\n":) wsp;                                          9
};                                                                                10

POpt wsp_crlf_t o_wsp_crlf_t;                                                     12

Pstruct lws_t {                                                                   14
   o_wsp_crlf_t wsp_crlf;                                                         15
   PString_ME(:"(\\s|\\t)+":) wsp;                                                16
};                                                                                17

POpt lws_t sws_t;                                                                 19

Pstruct hcolon_t {                                                                21
   PString_ME(:"(\\s|\\t)*":) sp_or_htab;                                         22
   ':';   sws_t   sws;                                                            23
};                                                                                24

Pstruct CSeq_t {                                                                  26
   PString_ME(:"[Cc][Ss][Ee][Qq]":) name;                                         27
   hcolon_t hcolon;                                                               28
   CSeq_number_t number;                                                          29
   lws_t lws;                                                                     30
   method_t method;                                                               31
};                                                                                32

Precord Pstruct SIP_msg {                                                         34
   request_line_t request_line;                                                   35
   [...]
   CSeq_t cseq: checkCSeqMethod(request_line,cseq);                               37
   [...]
};                                                                                39
```

Figure 2: PADS specification of the SIP RFC 3261

the same default parsing strategies as ABNF, and thus *e.g.*, case insensitive strings must be specified explicitly using regular expressions (line 27). Similarly, translating SIP whitespace into PADS requires writing many lines of specifications (lines 8-19), including regular expressions. 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 to 6), which are used here in the case of the CSeq number (line 29), 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 to 2 and line 37).

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 [1]. While a tool has been developed to convert an ABNF specification to a regular expression [1], 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 constraints specified by the RFC.

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 information in the required header fields incur the time cost of parsing this information and the space cost of storing the result (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 on his own to develop complex parsing code.

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 built 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, as we show in 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 [2], 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.

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. Thus, 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

```
1  message sip3261 {
2      request {
3          ; Request only
4          requestLine = Method:method SP  Request-URI:uri  SP SIP-Version

6          ; Constraints that apply only for the CSeq of a request
7          header CSeq { CSeq.method == requestLine.method }

9          ; Constraints that apply only for the Max-Forwards of a request
10         header Max-Forwards { mandatory }

12     }

14     response {
15         ; Response only
16         statusLine  = SIP-Version SP Status-Code:code SP Reason-Phrase:rphrase
17         [...]
18     }

20     enum Method      =  INVITEm / ACKm / OPTIONSm / BYEm / CANCELm / REGISTERm / extension-method
21     extension-method  =  token
22     INVITEm           =  %x49.4E.56.49.54.45 ; INVITE in caps
23     [...]

25     struct Request-URI =  SIP-URI / SIPS-URI / absoluteURI { lazy }
26     [...]

28     uint16 Status-Code =  Informational / Redirection / Success / Client-Error / Server-Error
29                           / Global-Failure / extension-code
30     uint16 Global-Failure  =      "600"  ;  Busy Everywhere
31                           /  "603"  ;  Decline
32                           /  "604"  ;  Does not exist anywhere
33                           /  "606"  ;  Not Acceptable
34     uint16 extension-code  =  3DIGIT { extension-code >= 100 && extension-code <= 699 }
35     [...]

37     ; Header CSeq
38     header CSeq = 1*DIGIT:number as uint32 LWS Method:method

40     ; Header Max-Forwards
41     header Max-Forwards = 1*DIGIT:value as uint32 { mandatory }

43     ; Header To
44     header To { "to" / "t" } = ( name-addr / addr-spec:uri ) *( SEMI to-param ) { mandatory
45     name-addr        =  [ display-name ] LAQUOT addr-spec:uri RAQUOT
46     struct addr-spec =  SIP-URI / SIPS-URI / absoluteURI { lazy }

48     [...]
49 }
```

Figure 3: Excerpt of the Zebu Specification for the SIP protocol

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.

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 (*c.f.*, Figure 3, line 4):

```
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 10 of Figure 1 is reorganized into the following Zebu rule (*c.f.*, Figure 3, line 38)

```
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 44 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 in a message (`multiple`). For example, in the SIP specification, the header `To` is specified to be mandatory and read-only (line 44). 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 7.

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 2-12) indicates that for the `CSeq` header the method must be the same as the method in the request line (line 7), and that the `Max-Forwards` header is mandatory (line 10). The constraints in the response block (lines 14-18) 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 4, 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 in the message text. This is the case of `method` and `uri` in our example. 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 38 the CSeq number is specified as being a `uint32`. Nonterminals can also be specified as structures (`struct`), unions (`union`), and enumerations (`enum`). A structure collects all derived named subfields. As illustrated in the case of `Request-URI` (line 25), a structure may even be used in the case of an alternation, when the application does not need to know from what element of the alternation a named entry is derived. A union, in contrast, records which alternation was matched and in each case only includes subfields derived from the given alternation. Finally, an enumeration is a special case of union in which the only information that is recorded is the identity of the matching alternation; the matched data is not stored. In line 20, for example, `Method` is specified as being an enumeration, because the application only needs to know whether the method of the message is one of the standard ones or an extension method, but does not need to know the identity of the extension method in the latter case.

An application may use the information in certain subfields only in some exceptional cases. The Zebu constraint `lazy` allows the programmer to specify that a specific subfield should not be parsed until requested by the application. For example, in the SIP specification, `Request-URI` has this annotation (line 25).

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

```
// Init
extern sip3261           sip3261_init();
// Top level parser
extern void              sip3261_parse(sip3261, char *, int);
// Generic parser for headers
extern void              sip3261_parse_headers(sip3261, E_Headers);
// Dedicated parser for addr-spec
extern void              sip3261_parse_addr_spec(T_Lazy_addr_spec);
// Accessors
extern T_bool            sip3261_isRequest(sip3261);
extern T_bool            sip3261_isResponse(sip3261);
extern T_RequestLine     sip3261_get_RequestLine(sip3261);
extern T_header_From     sip3261_get_header_From(sip3261);
extern T_Method          sip3261_RequestLine_getMethod(T_RequestLine);
extern T_MethodEnum      sip3261_Method_getType(T_Method);
extern T_Str             sip3261_Method_getValue(T_Method);
extern T_Lazy_addr_spec  sip3261_header_From_getUri(T_header_From);
extern T_addr_spec       sip3261_Lazy_Addr_spec_getParsed(T_Lazy_addr_spec);
extern T_Option_Str      sip3261_Addr_spec_gethost(T_addr_spec);
extern T_Str             sip3261_Option_Str_getVal(T_Option_Str);
[...]
```

4a. Generated stubs

```
sip3261 msg = sip3261_init();
sip3261 msg = sip3261_parse(msg, buf, len);                                                    1
// Process only request messages                                                               2
if (sip3261_isRequest(msg)) {                                                                  3
  // Filter INVITE methods                                                                     4
  T_RequestLine requestLine = sip3261_get_RequestLine(msg);                                    5
  if (sip3261_Method_getType(sip3261_RequestLine_getMethod(requestLine)) == E_INVITEm) {       6
    // We parse only the header From                                                           7
    sip3261_parse_headers(msg, E_HEADER_FROM);                                                 8
    T_Lazy_addr_spec l_addr_spec = sip3261_header_From_getUri(sip3261_get_header_From(msg));   9
    sip3261_parse_addr_spec(l_addr_spec);                                                     10
    T_Option_Str host = sip3261_Lazy_Addr_spec_getParsed(l_addr_spec);                        11
    // host may be undefined in some cases, check it and log its value                        12
    if (sip3261_Option_Str_isDefined(host)) {                                                 13
       mylog(sip3261_Option_Str_getVal(host));                                                14
}}}                                                                                           15
```

4b. Application logic

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

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, whether done by hand or using a parser generator. The Zebu compiler thus 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 30, 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 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 names of these stub functions depend on the specific structure of the grammar, but follow a well-defined schema that facilitates their use by the application developer.

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 [12] library for matching the regular expression of a header value

that has been derived from the ABNF specification. The parsing functions 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 6). If so, it uses the function `sip3261_parse_headers` to parse the `From` header field (line 8), and then the functions `sip3261_header_From_getUri` and `sip3261_get_header_From` to extract the URI (line 9). Line 46 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 10). After a check that the host name is present (line 13), its value is extracted using the function `sip3261_Option_Str_getVal` in line 14.

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 network messages, to avoid corrupting its internal state. As the parser is the front-line in the treatment of network messages, it has a key role to play in providing this robustness. In this section, we evaluate the robustness improvement offered by Zebu, by comparing the reaction of Zebu-based parsers and a variety of existing parsers to valid and invalid network messages. Our experiments are based on a mutation analysis technique.

For SIP, we compare with the oSIP and SER parsers previously described in Section 2. For RTSP, we use the parser in the widely used VLC media player and streaming server [32], and the parser provided by the LiveMedia library [1]. Figure 5 shows the sizes of the ABNF and Zebu specifications of the message grammars for SIP and RTSP. The Zebu specification is longer, because it includes rules that are mentioned only by reference to another RFC in the original SIP and RTSP specifications. Figure 5 also shows the number of lines of code in the oSIP, SER, VLC, and LiveMedia parser implementations.

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

Figure 5: The sizes of the SIP and RTSP message grammars, and the sizes of existing parsers. Sizes in lines of code.

In the rest of this section, we first introduce mutation analysis, and then compare the robustness of existing SIP and RTSP parsers with that of the corresponding Zebu-based parsers. Finally, we evaluate the performance of Zebu, showing that Zebu-based parsers are often as efficient as hand-written ones.

4.1 Robustness evaluation

Mutation analysis is a fault-based testing technique for unit-level testing [7]. Traditional mutation testing involves introducing small changes, *i.e.*, mutations, in program source code, to determine whether a given test suite is sufficient to distinguish between correct and incorrect programs. In our case, however, we are interested

[1] LiveMedia: Streaming Media, `http://www.livemediacast.net/`

in assessing the robustness of the program, *i.e.*, the parser, and thus we introduce mutations into the test data, *i.e.*, the network messages, rather than into the program source code. We use mutation rules both to generate invalid messages and to generate valid messages that have properties that are known to be challenging for network protocol message parsers. A robust parser should reject the invalid messages and accept the valid ones.

To generate invalid messages, we have defined a set of mutation rules for messages based on ABNF structure:

- Mutations on the characters set. Message literals are derived from a fixed set of possible characters. The first, middle, or last character of a message literal is replaced with any character outside the valid set.
- Mutations on repetitions. As described in Section 2.1, ABNF offers a generic mechanism of repetition. Mutants are chosen to describe an invalid number of repetitions.
- Mutations based on constraints. Protocol specifications include additional constraints not specified in the message grammar about the values of header subfields. For example, the response code of a SIP response is not only an unsigned integer of three digits, but its value must also be less than 699 (see Figure 3). Mutants are chosen that violate these constraints.

To generate valid but problematic messages, we have extended our character set mutation rule to create messages of the form suggested by the SIP Torture Test Message RFC [28]. This RFC describes a set of valid SIP messages that test corner cases in SIP implementations.

To compare the robustness of Zebu-based applications to applications based on hand-crafted parsers, we consider the parsing of the principal fields of a network protocol message. For SIP, these fields are the command line and the six mandatory header fields, while for RTSP they are the command line and the header fields `Transport`, `CSeq` and `UserAgent`. We drive each of the parsers listed in Figure 5 using minimal applications that request access to these fields. 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 log statistical information about incoming messages. These applications use the stubs generated by the Zebu compiler to access network messages, analogous to the code illustrated in Figure 4b. 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.

Invalid messages In our first set of tests, we apply our mutation rules to SIP and RTSP messages, generating a stream of invalid messages, which we then send to each of the SIP and RTSP applications, respectively. As shown in Figure 6, while the Zebu-based applications detect every mutant as representing an invalid message, none of the hand-crafted parsers detects more than about 25% of the injected mutants. This situation may have a critical impact. In the case of SIP for example, 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.

		Mutation sites	Injected Mutants	Detected Mutants	% detected Mutants
SIP	`log-Zebu-SIP`	81	5976	5976	**100.0**%
	`log-oSIP`			1020	**17.1**%
	`log-SER`			1512	**25.3**%
RTSP	`log-Zebu-RTSP`	19	2730	2730	**100.0**%
	`log-VLC`			4	**0.1**%
	`log-LiveMedia`			748	**27.4**%

Figure 6: Mutation coverage for invalidSIP and RTSP messages

Valid messages While message parsers should detect erroneous messages as early as possible to preserve the robustness of the applications that use them, they also must correctly parse valid messages. The SIP Torture Test Message RFC [28] describes a set of valid SIP messages that test corner cases in SIP implementations. Guided by this RFC, we have extended our character set mutation rule to generate mutants that are valid SIP messages but are designed to torture a SIP implementation. Figure 7 shows that up to about 4% of the valid messages are rejected by hand-crafted SIP parsers. By comparison, the Zebu-based SIP parser strictly follows the message grammar.

We have tried 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 mutated messages.

		Mutation sites	Injected Mutants	Rejected Mutants	% rejected Mutants
SIP	log-Zebu-SIP	18	549	0	**0.0**%
	log-oSIP			21	**3.9**%
	log-SER			2	**0.4**%

Figure 7: Mutation coverage for valid SIPmessages

4.2 Performance Evaluation

We now compare the performance of Zebu-based parsers to that of hand-crafted ones. Our results are only for SIP, which is the most demanding in terms of performance. For our experiments, we have implemented four versions of the SIP message statistics reporting application described in Section 3.4. In each case, the application records the host information of the URI stored in the `From` header field of an `INVITE` message. The first version (`inv-SER-module`) is implemented as a dedicated SER module to obtain full access to the internal data structures of SER. The second version (`inv-SER-exec`) is written using the configuration language of SER and relies on the escape mechanism provided by SER to invoke `sed` to extract the host information, as described in Section 2.4. The third version (`inv-oSIP`) is implemented using a few lines of C code on top of the oSIP SIP stack. The last version (`inv-Zebu`) is the Zebu-based application depicted in Figure 4b.

Our application illustrates the case where an application such as a intrusion detection system needs to access a fragment of a header subfield. To explore the effect that various kinds of messages have on the parsing performance for such an application, we consider a collection of `INVITE` messages, which are relevant to our application, and an example of a non-`INVITE` message, which is not. Among the `INVITE` messages, in $\texttt{INVITE}_1$ the `From` header field contains only the URI subfield and an required tag subfield; all of the other subfields, which are optional, are omitted. This entails the minimal processing for a message that is relevant to the application. The remaining `INVITE` messages, $\texttt{INVITE}_2$ and $\texttt{INVITE}_3$, show the effect of varying the position of the `From` header field. In $\texttt{INVITE}_2$, the `From` header field is the first of 34 header fields, while in $\texttt{INVITE}_2$ it is the last of 34 header fields. The non-`INVITE` message is a `BYE` and has 7 headers.

		inv-SER-module		inv-SER-exec		inv-oSIP		inv-Zebu	
	Message size	**Cycles**	**Ratio**	**Cycles**	**Ratio**	**Cycles**	**Ratio**	**Cycles**	**Ratio**
$\texttt{INVITE}_1$	697	13 788	1	7 593 550	551	182 703	13	51 054	4
$\texttt{INVITE}_2$	1 734	13 595	1	8 803 456	648	276 275	20	80 270	6
$\texttt{INVITE}_3$	1 734	32 045	1	10 015 827	313	-	-	133 164	4
BYE	334	10 252	1	10 765	1	105 773	10	6 037	0.6

Figure 8: Performance of SIP applications (time in cycles, ratio as compared to `inv-SER-module`)

Our experiments were performed using a Pentium III (1GHz) as the server, which is stressed by a bi-processor Xeon 3.2Ghz client. Figure 8 compares the parsing time for each of the applications to that of `SER-module`, which has the fastest parser among the existing parsers that we tested.

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 `inv-SER-module` is particularly efficient in the case of `INVITE` messages, as the information required by the application is already available in the SER internal data structures. The parsing done by `inv-SER-exec` is roughly as efficient as that done by `inv-SER-module` for the non-`INVITE` message. The parsing done by `inv-SER-exec` for the `INVITE` messages, on the other hand, is up to 648 times slower, because it forks a `sed` process. Despite the bad performance in this case, the use of the configuration language of SER remains relevant, because it provides ease of programming and safety, which are not provided by the use of a SER module.

The parsing done by `inv-oSIP` is over 13 times slower than the parsing done by `inv-SER-module` for `INVITE` messages and over 10 times slower for the non-`INVITE` 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 the subfields into a data structure that is provided to the application, roughly doubling the execution time. No results are presented for `inv-oSIP` for $\texttt{INVITE}_3$, because oSIP crashes on this message.

Finally, while Zebu follows the same two-level parsing strategy as SER, the parsing done by `inv-Zebu` is significantly slower than the parsing done by `inv-SER-module` for the `INVITE` messages, because Zebu checks the URI more rigorously than SER. On the other hand, Zebu is significantly more efficient than SER for the non-`INVITE` 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 provides better

performance in such cases by being more closely tailored to the needs of the application, and retains safety, which is lost in SER when using the module approach.

5 Related Work

Parser generators such as DATASCRIPT [3], PacketTypes [18], PADS [9], GAPA [5] and binpac [22] have been recently developed to address the growing complexity of network protocol messages. However, as described in Section 2, these tools do not fulfill all the requirements of network application developers. APG [17] is a parser generator that accepts ABNF directly. Semantic actions are specified via callback functions rather than annotations on the grammar. We have found the use of such callback functions to be somewhat heavyweight, in our experience in using 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, which we have found (Section 4) essential to obtaining good performance.

Domain-specific languages have been used successfully in various application domains including operating systems [16, 19] and networks [11, 13]. Several of these languages have explicitly targeted improving system robustness. The Devil language, 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 [25]. The language Promela++ for specifying network protocols, can be translated automatically both into the model checking language Promela [14] and into efficient C code, thus easing the development of a protocol implementation that is both verified and efficient [4].

Mutation analysis has been used to test the robustness of other software components, such as operating systems [8], network intrusion detection systems [31], and databases [30]. Our work is most similar to the work on network intrusion detection systems, which also mutates network protocol messages.

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 particularly focused on analyzing the improvement in robustness that it provides. For this, we have defined a test methodology based on a mutation analysis that injects errors into network messages. We have applied our test methodology to SIP and RTSP servers by comparing existing parsers with Zebu-generated ones.

The results of 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 [23] via a stream of 2416 incorrect messages, sent within a space of 17 seconds. Because SER is used for telephony, which is a critical service, 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 the added safety and robustness provided by Zebu does not significantly impact performance. In the case of SIP, in micro-benchmarks, we have found that a Zebu-based parser is often as efficient as a hand-crafted one.

References

[1] T. Akira. Converter from ABNF to regexp. `http://cvs.m17n.org/~akr/abnf/`.

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

[3] G. Back. DataScript - a specification and scripting language for binary data. In *Proceedings of the ACM Conference on Generative Programming and Component Engineering*, number 2487 in Lecture Notes in Commputer Science, pages 66–77, Pittsburgh, PA, USA, Oct. 2002.

[4] 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*, San Francisco, CA, USA, Mar. 1998.

[5] 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*, San Diego, CA, USA, Feb. 2007.

[6] D. Crocker and P. Overell. Augmented BNF for Syntax Specifications: ABNF. Internet Engineering Task Force: RFC 2234, Nov. 1997.

[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: help for the practicing programmer. *Computer*, 11(4):34–41, Apr. 1978.

[8] J. Duraes and H. Madeira. Characterization of operating systems behavior in the presence of faulty drivers through software fault emulation. In *2002 Pacific Rim International Symposium on Dependable Computing*, pages 201–209, Dec. 2002.

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

[10] Gaim 1.5.0: A multi-protocol instant messaging (IM) client. `http://gaim.sourceforge.net`.

[11] 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.

[12] P. Hazel. PCRE - perl compatible regular expressions. `http://www.pcre.org/`, July 2006.

[13] 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.

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

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

[16] 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.

[17] D. T. Lowell. APG - an ABNF parser generator. `http://www.coasttocoastresearch.com/`, June 2006.

[18] 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.

[19] 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 (OSDI 2000)*, pages 17–30, San Diego, California, Oct. 2000.

[20] A. Moizard. The GNU oSIP library. `http://www.gnu.org/software/osip`, June 2001.

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

[22] 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.

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

[24] 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 (OSDI 2002)*, pages 45–60, Boston, MA, USA, Dec. 2002.

[25] 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, Göteborg, Sweden, July 2001. IEEE Computer Society.

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

[27] rtsp.org: Real time streaming protocol information and updates. `http://www.rtsp.org`.

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

[29] A. Spencer. Asterisk: The open source PBX. `http://www.asterisk.org`.

[30] J. Tuya, M. J. Suárez-Cabala, and C. de la Riva. Mutating database queries. *Information and Software Technology*, 2006. To appear.

[31] G. Vigna, W. Robertson, and D. Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 21–30, Washingtion, DC, USA, Oct. 2004.

[32] The videolan project. `http://www.videolan.org/vlc/`.

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399