Parsifal: a pragmatic solution to the binary parsing problem

Olivier Levillain
Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI)
https://github.com/ANSSI-FR
firstname.lastname@ssi.gouv.fr

***Abstract*—Parsers are pervasive software basic blocks: as soon as a program needs to communicate with another program or to read a file, a parser is involved. However, writing robust parsers can be difficult, as is revealed by the amount of bugs and vulnerabilities related to programming errors in parsers. It is especially true for network analysis tools, which led the network and protocols laboratory of the French Network and Information Security Agency (ANSSI) to write custom tools. One of them, Parsifal, is a generic framework to describe parsers in OCaml, and gave us some insight into binary formats and parsers. After describing our tool, this article presents some use cases and lessons we learned about format complexity, parser robustness and the role the language used played.**

In 2010, the Electronic Frontier Foundation scanned the Internet to find out how servers answered on the 443/TCP port worldwide [EFF12], [EB10a], [EB10b]. We studied this significant amount of data with custom tools, to gain thorough insight of the data collected [LEDM12]. However, the data contained legitimate TLS [DR08] messages, as well as invalid messages or even messages related to other protocols (HTTP, SSH). To face this challenge and extract relevant information, we needed robust tools on which we could depend. Yet, existing TLS stacks did not fit our needs: they can be limited (some valid options are rejected), laxist (they silently accept wrong parameters) or fragile (they crash on unexpected values, even licit ones). Thus, we decided to write our own parsers.

Our first attempt to write a TLS parser was with the Python language; it was quickly written and allowed us to extract some information. However, this implementation was unacceptedly slow. The second parser was in C++, using templates and object-oriented programming; its goal was to be flexible and fast. Yet, the code was hard to debug (memory leaks, segmentation faults on flawed inputs), and lacked extensibility since every evolution of the parsers required many lines of code.

So a new version was written, in OCaml: it used a DSL (Domain Specific Language) close to Python to describe the structures to be studied. This third parser was as fast as the previous one, less error-prone, but still needed a lot of lines to code simple features. That is why we finally decided to use a preprocessor to do most of the tedious work, letting the programmer deal only with what's important, structure description, while avoiding usual mistakes in low-level memory management. This last implementation, Parsifal, has all the properties we expected: efficient and robust parsers, written using few lines of code.

Our work originally covered X.509 certificates and TLS messages, but we soon tried Parsifal on other network protocols (BGP/MRT, DNS, TCP/IP stack, Kerberos) and on some file formats (TAR, PE, PCAP, PNG). Some of these parsers are still at an early stage, but one of the strength of Parsifal is that it is easy to describe part of a protocol, and focus only on what really needs to be dissected.

Sec. I succinctly presents how Parsifal works. Then, we describe how to write a PNG parser using Parsifal in Sec. II. Next, Sec. III presents some difficulties related to binary parsers through case studies, to emphasize the properties of Parsifal-based parsers: robustness, conciseness, expressiveness. Sec. IV presents some elements of comparison between standard tools and Parsifal for several formats described. Sec. V presents the lessons learned by spending years writing parsers. Finally, Sec. VI compares our work to existing tools.

I. Parsifal: a quick tour

A. PTypes

Parsifal relies on the idea that tedious code should not be written by humans since it can be generated. The basic blocks of a Parsifal parser are PTypes, that is OCaml types augmented by the presence of some manipulation functions: a PType `t` is composed of:

- the corresponding OCaml type `t`;
- a `parse_t` function, to transform a binary representation of an object into the type `t`;
- a `dump_t` function, that does the reverse operation, that is dumping a binary representation out of a constructed type `t`;
- a `value_of_t` function, to translate a constructed type `t` into an abstract representation, which can then be printed, exported as JSON, or analysed using generic functions.

There are essentially three kinds of PTypes:

- basic PTypes, provided by the standard library, like integers, strings, lists or ASN.1 DER basic objects;
- keyword-assisted PTypes, like structures, are descriptions using a pseudo-DSL integrated to the language thanks to a preprocessor (some of the offered constructions are presented in the remaining of the section);
- custom PTypes: when the corresponding structure is too complex to simply describe, you can always go back to manually writing the type and the functions.

DOI 10.1109/SPW.2014.35

B. Examples of construction

Among the TLS messages, TLS alerts are used to signal a problem during the session. Such messages are simply composed of an alert level (one byte with two possible values) and an alert type (another byte). Here is the corresponding extract of the *specification* [DR08]:

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
    close_notify(0),
     ...
    unsupported_extension(110),
    (255)
} AlertDescription;

struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```

Code snippet 1.

It is possible to describe such messages in Parsifal with the following code:

```
enum alert_level (8, UnknownVal AL_Unknown) =
  | 1 -> AL_Warning
  | 2 -> AL_Fatal

enum alert_type (8, UnknownVal AT_Unknown) =
  | 0 -> AT_CloseNotify
  ...
  | 110 -> AT_UnknownExtension

struct alert = {
  alert_level : alert_level;
  alert_type : alert_type
}
```

Code snippet 2.

As a result, the preprocessor will generate three OCaml types, and some functions:

```
(* alert_level *)

type alert_level =
    AL_Warning
  | AL_Fatal
  | AL_Unknown of int

(* parse/dump/value_of functions *)
val parse_alert_level : input -> alert_level
val dump_alert_level : output -> alert_level -> unit
val value_of_alert_level : alert_level -> value


(* alert_type *)

type alert_type =
    AT_CloseNotify
  ...
  | AT_Unknown of int

(* ... 3 functions, similar to those relative *)
(*             to alert_level ...             *)

(* alert *)

type alert = {
  alert_level : alert_level;
  alert_type : alert_type;
}
val parse_alert : input -> alert
val dump_alert : output -> alert -> unit
val value_of_alert : alert -> value
```

Code snippet 3.

The constructions available in Parsifal are enumerations (`enum` keyword), records (`struct`), choices allowing for types depending on a parameter (`union`), ASN.1 DER structures and choices (`asn1_struct` and `asn1_union`) and aliases (`alias` and `asn1_alias`).

II. A COMPLETE EXAMPLE: A BASIC PNG PARSER

As an example of the conciseness of Parsifal code, this section briefly describes how to write a simple PNG parser. A PNG image is composed of a magic number (`"\x89PNG\r\n\x1a\n"`) followed by a list of chunks, which are described in Table I.

Offset	Field	Size	Type
0	Chunk size `sz`	4	Big-endian integer
4	Chunk type	4	String
8	Chunk data	`sz`	Chunk-dependent
8 + `sz`	CRC	4	CRC 32

TABLE I.

Using this first definition of the PNG format, it is easy to write some code in Parsifal:

```
struct png_chunk = {
  chunk_size : uint32;
  chunk_type : string(4);
  chunk_data : binstring(chunk_size);
  chunk_crc : uint32;
}

struct png_file = {
  png_magic : magic("\x89\x50\x4e\x47\x0d\x0a\x1a\x0a");
  chunks : list of png_chunk;
}
```

Code snippet 4.

The first `struct` definition describes what a chunk is, and the `png_file` one what a PNG file is. Since Parsifal automatically generates the associated `parse`, `dump` and `value_of` functions, a complete tool extracting PNG chunks can be written by adding some lines:

```
let input = string_input_of_filename Sys.argv.(1) in
let png_file = parse_png_file input in
print_endline (print_value (value_of_png_file png_file))
```

Code snippet 5.

Here is the result of the compiled programs on a PNG file:

```
value {
  png_magic: 89504e470d0a1a0a (8 bytes)
  chunks {
    chunks[0] {
      chunk_size: 13 (0x0000000d)
      chunk_type: "IHDR" (4 bytes)
      chunk_data: 000000140000001604030000000 (13 bytes)
      chunk_crc: 846176565 (0x326fa135)
    }
    chunks[1] {
      chunk_size: 15 (0x0000000f)
      chunk_type: "PLTE" (4 bytes)
      chunk_data: ccfffffcc99996633333333000000 (15 bytes
     )
      chunk_crc: 1128124197 (0x433dcf25)
    }
    chunks[2] {
      chunk_size: 1 (0x00000001)
      chunk_type: "bKGD" (4 bytes)
      chunk_data: 04 (1 bytes)
      chunk_crc: 2406013265 (0x8f68d951)
    }
    chunks[3] {
      chunk_size: 77 (0x0000004d)
      chunk_type: "IDAT" (4 bytes)
      chunk_data: 78da63602005b8000184c5220804... (77
     bytes)
```

```
      chunk_crc: 466798482 (0x1bd2c792)
    }
    chunks[4] {
      chunk_size: 86 (0x00000056)
      chunk_type: "tEXt" (4 bytes)
      chunk_data: 436f6d6d656e7400546869732061... (86
     bytes)
      chunk_crc: 1290335428 (0x4ce8f4c4)
    }
    chunks[5] {
      chunk_size: 0 (0x00000000)
      chunk_type: "IEND" (4 bytes)
      chunk_data: "" (0 byte)
      chunk_crc: 2923585666 (0xae426082)
    }
  }
}
```

Code snippet 6.

Of course, this is only the beginning, and one would likely want to improve the description by enriching the chunk content. This is actually a strength of our methodology: it is generally easy to describe the big picture and then to refine the parser to take into account more details.

A. Union and progressive enrichment

To illustrate how to enrich the chunk data, let us start with the image header, corresponding to the `"IHDR"` type. It is supposed to contain the following structure:

```
struct image_header = {
  width : uint32;
  height : uint32;
  bit_depth : uint8;
  color_type : uint8;
  compression_method : uint8;
  filter_method : uint8;
  interlace_method : uint8;
}
```

Code snippet 7.

To use this new structure when dealing with a `"IHDR"` chunk, we have to create a union PType, depending on the chunk type:

```
union chunk_content [enrich] (UnparsedChunkContent) =
| "IHDR" -> ImageHeader of image_header
```

Code snippet 8.

Finally, we have to rewrite the `chunk_data` field in the `png_chunk` structure:

```
struct png_chunk = {
  chunk_size : uint32;
  chunk_type : string(4);
  chunk_data : container(chunk_size) of
                 chunk_content(chunk_type);
  chunk_crc : uint32;
}
```

Code snippet 9.

It should be clear how to enrich other chunk types from now: write the PType corresponding to the chunk content, and then add a line in the `chunk_content union`. When facing an unknown chunk type, `parse_chunk_content` will produce an `UnparsedChunkContent` value containing the unparsed string.

B. PContainers: a useful concept

As we began using Parsifal to describe various file formats and network protocols, it dawned on us that it might be useful to create another concept that could be reused: PContainers. Initially, the only existing containers were lists, but the notion of containers can be broader: the abstraction of a container containing a PType makes it possible to automate some processing that has to be done at parsing and/or dumping time. Here are some examples:

- encoding: hexadecimal, base64;
- compression: DEFLATE, zLib or gzip containers;
- safe parsing: some containers provide a fall-back strategy when the contained PType can not be parsed;
- miscellaneous checks: CRC containers are useful to check a CRC at parsing time and to generate the CRC value at dumping time.

There again, the idea is to reuse code to reduce the time spent writing the same code time and again. Here is an example of an ancillary PNG chunk called `iCCP` (Embedded Profile), which contains a null-terminated string (called `cstring`) that should not exceed 80 characters, a byte field and a compressed string. Using standard Parsifal containers, the chunk can then be described as follows:

```
struct embedded_profile = {
  name : length_constrained_container(AtMost 80) of
           cstring;
  compression_method : uint8;
  compress : ZLib.zlib_container of string;
}
```

Code snippet 10.

C. Custom PTypes

Finally, when it is not possible to describe a PType using keywords like `struct` or `union`, it is always possible to write a PType from scratch.

Assuming it does not exist already in the standard library, here is how you could describe the null-terminated string `cstring` as a custom PType. The *intended* type is a simple string, but the corresponding `parse` and `dump` functions have to be written manually:

```
type cstring = string

let rec parse_cstring input =
  let next_char = parse_char input in
  if next_char <> '\x00'
  then (String.make 1 next_char) ^ (parse_cstring input)
  else ""

let dump_cstring buf s =
  POutput.add_string buf s;
  POutput.add_char buf '\x00'
```

Code snippet 11.

Another example of custom PType in the PNG description is the chunk itself. With the `struct` definition presented earlier, it is possible to create and dump inconsistent chunks where the length or the CRC do not correspond to the data contained. This might be useful for fuzzing purpose, but it makes it harder to produce valid values:

```
let data_chunk = {
  chunk_size = String.length png_data;
  chunk_type = "IDAT";
  chunk_data = UnparsedChunkContent png_data;
  chunk_crc = Crc.crc32 ("IDAT" ^ png_data);
}
```

Code snippet 12.

An assumed design choice in Parsifal is to simplify the manipulation (parsing and dumping) of valid files when possible, even if it makes other use cases (like fuzzing) less accessible. The following snippets present a custom PType to replace `png_chunk` (only the parse function is detailed) and how the new, simpler, way to create a chunk:

```
type png_chunk = {
  chunk_type : string;
  chunk_data : chunk_content;
}

let parse_png_chunk input =
  let size = parse_uint32 input in
  let raw_data = peek_string (size + 4) input in
  let chunk_type = parse_string 4 input in
  let data = parse_container size
               (parse_chunk_content chunk_type) input in
  let crc = parse_string 4 input in
  let computed_crc = Crc.crc32 raw_data in
  if computed_crc <> crc then failwith "Invalid CRC";
  { chunk_type = chunk_type; chunk_data = data }
```

Code snippet 13.

```
let data_chunk = {
  chunk_type = "IDAT";
  chunk_data = UnparsedChunkContent png_data;
}
```

Code snippet 14.

The resulting variable can then be easily integrated in a chunk list to produce a PNG file.

III. Case studies

For the moment, our main goal has been to write robust parsers to analysing data but, most importantly, to understand how some protocols or file formats really work. The initial parsers covered TLS messages and X.509 certificates, but we have been describing more and more formats, sometimes to study new areas, sometimes as a challenge to test Parsifal's expressivity. At the beginning, these challenges required changes in Parsifal preprocessor or standard library, but such modifications have become rare lately, which means we have reached a balance between language expressivity and implementation complexity. Here are some examples of encountered difficulties, as well as how we handled them in Parsifal.

A. ASN.1 RSA key

ASN.1 is usually considered complex to parse. However, it is important to remember that ASN.1 is an *abstract* syntax notation, that can be encoded using different rules. The Basic Encoding Rules (BER) allow variable length data structures, whereas the Distinguished Encoding Rules (DER) are more restrictive and require data are in canonical forms. For our needs, we only considered the latter, which uses a rather simple TLV (Tag/Length/Value) header.

To simplify the development of ASN.1 DER types, header and scalar value parsing is automated by Parsifal, through basic DER types implemented in Parsifal standard library, and thanks to new helper keywords to build ASN.1 sequences. For example, PKCS #1 [JK03] describes an ASN.1 structure for RSA private keys. The following code implements this type and is a complete program to parse a PEM[1] file named `key.pem` containing an RSA private key and print the corresponding modulus:

```
asn1_struct rsa_key = {
  version : der_smallint;
  modulus : der_integer;
  publicExponent : der_integer;
  privateExponent : der_integer;
  prime1 : der_integer;
  prime2 : der_integer;
  exponent1 : der_integer;
  exponent2 : der_integer;
  coefficient : der_integer
}

let input = string_input_of_filename "key.pem" in
let key = parse_base64_container parse_rsa_key input in
print_endline (hexdump key.modulus)
```

Code snippet 15.

B. DNS

At first, analysing DNS messages was a challenge to better understand the notion of *parsing context*. DNS resource records can indeed be *compressed* by referencing to previously read domain names. This form of compression requires a context retaining the domain names encountered during parsing. DNS parsing thus relies on the following data structure:

```
type dns_pcontext = {
  base_offset : int;
  direct_resolver : (int, domain) Hashtbl.t;
}
```

Code snippet 16.

The hash table is updated every time a new domain is parsed in the message, and can be used when encountering compressed domains. Offsets are computed relatively to the beginning of the message. The same effort must be done when dumping a message: record the offsets of the names produced and reuse them to compress the result.

C. DVI: an old format

More recently, we happen to look at the DVI file format, which proved to be an interesting anti-pattern, as far as binary formats are concerned. Indeed, a file can simply described by the following structures:

```
struct dvi_command = {
  opcode : opcode;
  command : dvi_command_detail (opcode);
}

alias dvi_file = list of dvi_command
```

Code snippet 17.

This means that a DVI file is a list of *commands*. Yet, command details (and in particular the details *length*) depend on the opcode. It is thus necessary, to parse a given file, to at least know the lengths of the DVI commands it contains. This clearly is *not* a desirable property, since it prevents a format from being extensible: adding new commands would probably break command alignment in older implementations, even if these commands are optional.

D. PE

To better understand UEFI, some co-workers had to study Portable Executable programs[2]. To this aim, they tried Parsifal, and had to deal with what strikes us as a bad idea: non-linear parsing. To analyse a PE, you have to walk through the file

[1] PEM is the name given to Base64-encoded DER data.

[2] Indeed, the format of Windows executables is used in UEFI.

using offsets, forwards and sometimes backwards. This is now possible in Parsifal, but it is very hard to check properties on such file formats.

IV. Performance comparison

This section proposes very partial comparisons between parser implementations. The criteria used are the number of lines needed to code the parser, and the time needed to parse a typical input.

A. Extracting certificates from TLS answers

First, let us compare several of our TLS parsers described in the introduction: the C++ parser, the OCaml one (using a DSL) and the Parsifal one. One typical tasks for those parsers was to extract the server certificates from an answer, when it was possible. The results on a typical input (containing both valid TLS messages and other random responses) are presented in Table II.

	C++	OCaml	Parsifal
LOC	8,5000	4,000	1,000
Processing time	100 s	40 s	8 s

TABLE II.

B. PNG parsers: the intern's choice

Two years ago, an intern studied some image file formats. Among the chosen formats, PNG was the first one to be developed. We instructed our intern to write two PNG parsers: one in C (a language he already knew) and one in OCaml using Parsifal[3]. Table III gives some elements to compare both implementations, tested against a set of 63,000 PNG images.

	C	Parsifal
LOC	4,000	350
Processing time	1,5 h	4 h

TABLE III.

C. MRT/BGP: the need for an efficient robust parser

To better understand the structure of internet and its resilience, we studied the BGP protocol (Border Gateway Protocol [RLH06]), through which networks are connected to form the internet. The RIPE[4] provides daily archives of BGP messages collected at different points of the internet, the collectors. This data, contained in MRT archives (Multi-Threaded Routing [BKL11]), amounts to about 2 GB per day.

At first, we used existing tools to extract the BGP route declarations. `libbgpdump`[5], a program written in C was the right tool, except when the program crashed due to complex memory bugs. Since we could not easily understand the reasons of the errors, we quickly rewrote a tool to parse MRT files and match the output of `libbgpdump`. This was done using Parsifal in three days. Table IV compares different implementations: the `libbgpdump` C version, an independent OCaml implementation, and the Parsifal-powered tool.

	`libbgpdump`	OCaml	Parsifal
LOC	4,000	1,200	550
Processing time	23 s	180 s	35 s

TABLE IV.

Both TLS and PNG cases compare different versions of the same code, written by the same person using different development methods. On the contrary, the MRT case shows results for implementations written by different people. Obviously the results only cover few use cases, but on these examples, Parsifal parsers are much shorter than the other ones. It is even harder to conclude anything about the relative speed of the compared tools; however, from our point of view, it is sometimes acceptable to pay the price for robustness, as in the MRT case.

V. Lessons learned

Parsifal has been developed at ANSSI for more than 3 years and its interface is becoming rather stable. In this section, we discuss several lessons we learned while writing binary parsers.

On formats

There exists such thing as a good or a bad format. Formats relying on simple *TLV* (Tag/Length/Value) structures are very easy to parse. Moreover, they allow for partial parsing (for example when considering unknown extensions). A concrete example of such a good format, according to our experience, is PNG: chunks respect the TLV logic and the corresponding structures are rather simple. However, as was shown with the `png_chunk` structure, a strict encapsulation of containers (length, CRC) could further simplify the parsing.

On the contrary, several properties leads to error-prone parsers and should be avoided. For example, *non-linear parsing* makes it unnatural to check whether the data we parse are in a licit location; this is the case in PE and JFIF formats, the latter actually being similar to a filesystem with directories and entries. Another undesirable property is when parsers are required to know every option to correctly interpret the file, such as the DVI format.

On the language

Parsifal was written in OCaml, a strongly-typed functional language. Here are the advantages of this language, regarding our goal to write parsers:

- the language is naturally expressive, which helps to write concise code;
- higher order functions allows to write containers easily (e.g. `parse_base64_container` naturally takes as an argument a parse function);

[3] As a side note, when the intern had to implement the JPEG format, he did not have enough time for two developments, and he chose Parsifal over C.

[4] The RIPE is one of five Region Internet Registries providing Internet resource allocations that support the operation of the Internet globally.

[5] http://www.ris.ripe.net/source/bgpdump

- as the memory is handled automatically by the garbage collector, several classes of attacks (double frees, buffer overflows) are impossible[6];
- pattern matching exhaustiveness check is a very helpful feature when dealing with complex structures or algorithms, since the compiler will remind you of unhandled cases.

On the process

In the end, our choice to automatically generate the tedious parts of the code paid: Parsifal allows to quickly write binary parsers, even for complex formats. What's more, the description process turned out to be accessible, even for persons with no initial OCaml (or functional programming) background.

However, you should not try to add everything in your DSL or in your constructions. Some parts of a parser are so complicated that they should remain manually written. That is the reason why we kept the possibility to fall back to manual PTypes when needed.

VI. Related work

Parsifal is not the first generic framework allowing for writing binary parsers. Popular alternatives are Scapy [BtSc12] and Hachoir [Sti12], two Python projects aiming respectively at dissecting network protocols and file formats. As our first target was TLS messages, we naturally tried to implement the protocol in Scapy. However, even if Scapy offers an expressive language to describe network packets, there were two major obstacles for our purpose: Python (and Scapy) is very slow, compared to other languages, which was not compatible with the amount of data to dissect; Python does not offer as many guarantees as OCaml (strong typing, exhaustive pattern-matching)[7]. As a side note, it is interesting to notice that Scapy's goal was not only to parse network packets, but also to forge such packets, even invalid ones. As was shown earlier, we chose a different path while developing Parsifal: robust and concise code instead of fuzzing features.

Another solution similar to Parsifal is the `binpac` language [RPRS06], developed within the Bro project. The idea is to describe network layers in a DSL, which is then compiled to produce C code. Here again, the C language does not offer the same guarantees as OCaml. What is more, some parsers we later wrote proved us that it was sometimes easier to manually write some parts of the parsers, which is not always simple with an external preprocessor, especially from the debugging point of view.

We also considered existing libraries in functional languages. For example, OCaml `Bitstring` library [Jon12] allows to parse bit fields by extending the pattern-matching, which can be very useful to parse binary protocols and formats. Moreover, `Bitstring` is also implemented through a `camlp4` preprocessor, like Parsifal. Yet, parsing bitstring is only a small part of what we needed, and Parsifal is able to automate many more cases, which could not really be implemented as `Bitstring` extensions.

Finally, another promissing functional language we considered is Haskell. Using typeclasses seemed to be a good way to automate code generation for parsing functions. As a matter, there exists a popular Haskell library, Parsec [Lei14]. Yet this library seemed essentially used to parse LL[1] grammars, which does not always fit binary formats. Furthermore, the automating process we really need is not easily available in Haskell: there is no standard way to automatically *derive* a function in Haskell for an arbitrary typeclass, which is exactly what Parsifal does.

VII. Conclusion

Parsifal is a generic framework to describe parsers in OCaml which has been used to describe several file formats and network protocols. From our point of view, this tool has all the expected properties: expressive language, code conciseness, efficient and robust programs. Moreover, Parsifal allows for an incremental description, which is useful to progressively learn the internals of a new format. The contribution of Parsifal to security is twofold. First it can help provide sound tools to analyse complex file formats or network protocols. Secondly we can implement robust detection/sanitization systems.

However, our tool has has several limits. First, due to some design choices, Parsifal is not meant to write fuzzers, but to simplify the parsing of *valid* inputs. Moreover, we currently only used Parsifal to analyse data and to build standalone sanitizing tools. One logical next step would be to use it in more real-world contexts, possibly with other programming languages, e.g. IDS software including our robust parsers. Future work also include writing libraries to completely handle file formats and network protocols, beyond the mere parsing step, e.g. a reference TLS stack or PNG tools.

Parsifal is publicly available under an open source license since June 2013 (https://github.com/ANSSI-FR/parsifal) and has been the subject of a tutorial during a conference last year [LDM13]. The git repository contains examples of step-by-step code description concerning TAR archives, the DNS protocol, PNG images and PKCS#10 CSR (Certificate Signing Request).

[6] However, this alone is far from perfect, since we may avoid arbitrary execution but not a fatal exception.

[7] ANSSI has led several works on the contribution of different programming languages to software security (see [JL14] in particular).

References

[BKL11] L. Blunk, M. Karir, and C. Labovitz. Multi-Threaded Routing Toolkit (MRT) Routing Information Export Format. RFC 6396 (Proposed Standard), October 2011.

[BtSc12] P. Biondi and the Scapy community. Scapy. http://www.secdev.org/projects/scapy/, 2003-2012.

[DR08] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[EB10a] P. Eckersley and J. Burns. An Observatory for the SSLiverse, Talk at Defcon 18, 2010.

[EB10b] P. Eckersley and J. Burns. Is the SSLiverse a safe place?, Talk at 27C3, 2010.

[EFF12] Electronic Frontier Foundation. The EFF SSL Observatory. https://www.eff.org/observatory, 2010-2012.

[JK03] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), February 2003.

[JL14] É. Jaeger and O. Levillain. Mind your languages: A discussion about languages and security. In *IEEE Security and Privacy LangSec workshop*, 2014.

[Jon12] R. Jones. bitstring. http://code.google.com/p/bitstring/, 2003-2012.

[LDM13] O. Levillain, H. Debar, and B. Morin. Parsifal: writing efficient and robust binary parsers, quickly. In *8th International Conference on Risks and Security of Internet and Systems (CRISIS)*, 2013.

[LEDM12] O. Levillain, A. Ebalard, H. Debar, and B. Morin. One Year of SSL Measurement. In *ACSAC*, 2012.

[Lei14] D. Leijen. Parsec, a fast combinator parser. http://hackage.haskell.org/package/parsec, 2001-2014.

[RLH06] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), January 2006. Updated by RFC 6286.

[RPRS06] Ruoming Pang and Robin Sommer. binpac: A yacc for Writing Application Protocol Parsers. In *Internet Measurement Conference*, 2006.

[Sti12] V. Stinner. Hachoir. https://bitbucket.org/haypo/hachoir/wiki/Home, 2009-2012.

APPENDIX

The current appendix describes Parsifal grammar, that is the rules corresponding to the constructions added to the OCaml language by our `camlp4` preprocessor.

A. Common tokens

To describe Parsifal constructions, we will use the following standard tokens:

- ⟨*ident*⟩ for a variable identifier;
- ⟨*constr*⟩ for a type constructor;
- ⟨*exception*⟩ for an exception constructor;
- ⟨*expr*⟩ for an arbitrary expression;
- ⟨*int_expr*⟩ for an expression representing an integer value;
- ⟨*int*⟩ for an integer literal.

Moreover, all Parsifal constructions can take options, which are defined by the following rules, where *LittleEndian* only relates to enumerations, while *EnrichByDefault* and *ExhaustiveChoices* are specific to unions. It is also possible to specify parameters to pass to `parse` and `dump` functions:

⟨*option*⟩ ::= LittleEndian
| EnrichByDefault
| ExhaustiveChoices
| ParseParam ⟨*ident*⟩
| DumpParam ⟨*ident*⟩

Another pervasive token in this grammar is ⟨*PType*⟩, which can either be an OCaml type, provided that the corresponding `parse` and `dump` functions exist, or a PContainer encapsulating a PType:

⟨*PType*⟩ ::= ⟨*ident*⟩ [(⟨*params*⟩)]
| ⟨*ident*⟩ [(⟨*params*⟩)] of ⟨*PType*⟩

Finally, for the sake of simplicity, a token named ⟨*foos*⟩ should be interpreted as a list of ⟨*foo*⟩ tokens. The corresponding rules have been omitted.

B. Enumerations

An enumeration is characterized by the size in bits of the underlying integers, the way to behave when facing an unknown value (throw an exception or use a fallback constructor), and a list of values (the enumeration cases):

⟨*enum_def*⟩ ::= enum ⟨*ident*⟩ [⟨*options*⟩] (⟨*int*⟩, ⟨*e_undef*⟩) = ⟨*e_cases*⟩

⟨*e_case*⟩ ::= ⟨*int_expr*⟩ → ⟨*constr*⟩
| ⟨*int_expr*⟩ .. ⟨*int_expr*⟩ → ⟨*constr*⟩

⟨*enum_undef*⟩ ::= Exception ⟨*exception*⟩
| UnknownVal ⟨*constr*⟩

C. Structures

A structure is essentially a sequence of fields, some of which can be optional, temporary fields (*checkpoints*) or made-up fields (*parse_field*, which do not relate to something in the binary input):

⟨*struct_def*⟩ ::= struct ⟨*ident*⟩ [⟨*options*⟩] = ⟨*s_fields*⟩

⟨*s_field*⟩ ::= [⟨*decorator*⟩] ⟨*ident*⟩ : PType

⟨*decorator*⟩ ::= optional
| parse_checkpoint
| parse_field
| dump_checkpoint

D. Unions

Unions is a list of union cases associating a discriminating value with a constructor, which usually contains a PType (in the absence of a PType, the constructed value is empty). In case the union can not be parsed, a default constructor is used. If no PType is specified for the default constructor, `binstring` is used:

⟨*union_def*⟩ ::= union ⟨*ident*⟩ [⟨*options*⟩] (⟨*constr*⟩ [of PType]) = ⟨*u_cases*⟩

⟨*u_case*⟩ ::= ⟨*expr*⟩ → ⟨*constr*⟩ [of PType]

E. Aliases

Some other constructions allow to describe PTypes: aliases to rename a PType, ASN.1 structures and aliases are syntactic sugar to describe an ASN.1 DER type (header and content):

⟨*alias_def*⟩ ::= alias ⟨*ident*⟩ [⟨*options*⟩] = PType

⟨*asn1_alias_def*⟩ ::= asn1_alias ⟨*ident*⟩ [⟨*options*⟩]
| asn1_alias ⟨*ident*⟩ [⟨*options*⟩] = ⟨*asn1_header*⟩ ⟨*PType*⟩

⟨*asn1_struct_def*⟩ ::= asn1_struct ⟨*ident*⟩ [⟨*options*⟩] = ⟨*s_fields*⟩

F. Parsifal standard PTypes

Here is a list of standard PTypes provided by the standard library:

- `uintX`: unsigned integers on X bits;
- `string` / `binstring`: character strings, which can span across a given number of bytes or the remaining of the input. The distinction between `string` and `binstring` is the way to represent the value once parsed;
- `bitbool` and `bitint` to describe bit fields;
- `magic` to handle magic values (expected markers);
- `ipv4` / `ipv6`;
- `cstring` for null-terminated strings
- `der_boolean`, `der_integer`, `der_bitstring`, `der_oid` and other basic ASN.1 types.

and here is a list of PContainers that are available in Parsifal:

- `list` / `array`: a list or an array of a given number of element. A list can also cover the remaining input;
- `base64_container` and `hex_container` to handle automatically these encoding;
- `zlib_container` to uncompress the input during parsing;
- `length_constraint`, for additional length checks;
- `safe_union`, to handle a failsafe mode in case a `parse` function fails with a union.