Work-in-Progress: Towards a Platform to Compare Binary Parser Generators

Olivier Levillain Sébastien Naud
Aina Toky Rasoamanana
Télécom SudParis
firstname.lastname@telecom-sudparis.eu

Abstract

Binary parsers are ubiquitous in the software we use everyday, be it to interpret file formats or network protocol messages. However, parsers are usually fragile and are a common place for bugs and security vulnerabilities.

Over the years, several projects have been developed to try and solve the problem, using different forms such as parser combinators or domain-specific languages. To better understand this rich ecosystem and offer tools to compare the existing solutions, we initiated a platform to test and compare such tools against different specifications.

Our so-called "LangSec Platform", which is a work-in-progress, aims at providing a framework to implement various specifications using different tools, which allow us to compare the expressiveness, the robustness and the efficiency of the included parser generators.

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. One way to solve the problem is to rely on tools to generate the actual parsers from high-level descriptions.

This approach can be effective, as soon as the target specification is not too much complex, the description language is expressive enough, and the parser generator produces robust and reliable code.

Over the years, many tools have been proposed to generate parsers, for different programming languages and with different flavors. Thus, we tried to analyze a subset of these tools to compare them and assess their relevance for different kinds of specifications. To this aim, we developed a platform to confront tools with different specifications.

Sec. 1 describes several existing parser generators that we found, and explains the choices we made to select several of them for a deeper study. Sec. 2 presents our platform and the specification we have started to implement with different tools. Sec. 3 contain our first results on the chosen tools and Sec. 4 gives a conclusion and some perspectives beyond this first step.

1 An Overview of the Parser Generator Landscape

In this section, we present a selection of parser generators. Table 1 presents several of their properties. In our study, the focus is mainly on binary formats, but several tools such as Hammer or Nom can also naturally handle textual formats.

Each tool is classified by its type: some tools provide a Domain-Specific Language (DSL) to describe the format to implement, whereas other rely on helper functions that are directly called from the programming language (a popular approach is to provide parser combinators).

We also provide some information about each project: the programming language they use and their public activity.

Tool	Type	Activity	Programming Languages
BinPAC [PPSP06]	DSL	2010-2021	C++
Hachoir	Helper Functions	2007-2021	Python
Haka	DSL	2013-2016	C, Lua
Hammer	**Parser Combinators**	**2012-2019**	**C, various bindings**
Kaitai Struct	**DSL**	**2016-2021**	**Scala, various bindings**
Nail [BZ14]	**DSL**	**2013-2015**	**C**
Netzob [Bos14]	**Helper Functions**	**2011-2020**	**Python**
Nom [Cou15]	**Parser Combinators**	**2014-2021**	**Rust**
Parsifal [Lev14]	**Embedded DSL**	**2011-2021**	**OCaml**
RecordFlux [RSCS19]	**DSL**	**2018-2021**	**Python, Ada**
Scapy	Helper Functions	2003-2021	Python
Spicy [SAH16]	DSL	2020-2021	C++

Table 1: Summary of the studied parser generators. Lines in boldface correspond to tools that were integrated in the platform at the time of writing.

Obviously, the presented list results is far from exhaustive[1], and is biased by our knowledge and experience. For example, having complex high-level network protocols such as TLS in mind does not lead to the same constructions as if one is looking at lower-level network protocols such as Ethernet or IPv4. Finally, it is worth noting one of the author of this paper is also the main developer of a tool in the list, Parsifal. Despite these biases, we believe trying to compare the tools has its merits, at least as a first step towards more objective analyses.

After a first study, we chose to only select several tools to focus on. Our idea was to obtain some diversity among our tool set; this is why we chose tools written in/for different programming languages, and using different paradigms. Table 1 summarizes the described tools and shows the ones we included in our platform.

When the study grows, we plan to integrate more tools to broaden the analyzed landscape.

2 The LangSec Platform

Our platform[2] is made of tools, specifications and implementations, and aims at studying different properties. Fig. 1 describes the platform and its components.

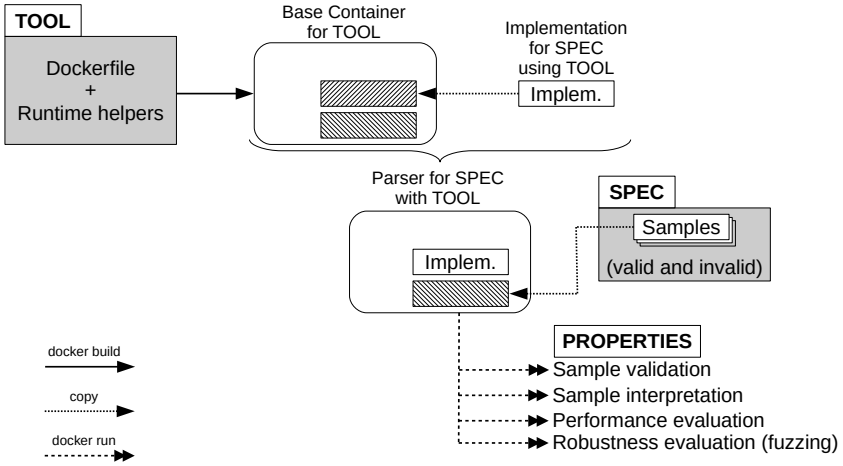

Figure 1: Platform Design.

[1] A longer list can be obtained on the `binary-parsing` GitHub repository at `https://github.com/dloss/binary-parsing`.

[2] The platform is available as a GitLab project: `https://gitlab.com/pictyeye/langsec-pf`.

A **tool** is a parser generator. The integration of a tool consists in writing a Dockerfile to build a container embedding the tool. Beyond merely compiling the project, each container aims at properly extracting the "implementation" part of a specification from the runtime required to make this implementation work. We thus only have to plug the corresponding hole to transform the generic container into a parser for a given specification.

A **specification** represents a file format or a network protocol, and is modeled by a set of good samples (that should be parsed properly) and bad samples (that should be rejected).

Currently, the specifications fall into two categories:

- the elementary constructions required to write binary parsers, such as a magic number, a list of 32-bit integers, a variable-length string, which allow to assess the basic expressiveness of each tool;
- more complex, realistic, specifications, such as DNS [Moc87], IP [Pos81b] or ICMP [Pos81a], which represent the real goal of the platform.

An **implementation** is a file written for a tool to recognize/parse samples conforming to a given specification.

Once we have these elements we can run the implementation for a given specification written for a given tool on the different samples for this specification, to study and compare the following **properties**:

- sample validation (which samples are accepted or rejected?);
- object interpretation (what was the actual result from the parsing operation?);
- performance (time, CPU and memory usage);
- implementation/tool robustness using a fuzzer.

For now, only the basic sample validation is functional for the 6 selected tools. Table 2 shows the status of the feature development.

	Hammer	Kaitai Struct	Nail	Netzob	Nom	Parsifal	RecordFlux
Sample Validation	✓	✓	✓	✓	✓	✓	✓
Object Interpretation						✓	
Performance measurement	Time	Time	Time	Time	Time	Time	Time
Fuzzing	✓		✓				

Table 2: Status of the feature development for each tool.

A concrete run on the platform can be seen on Fig. 2, as well as example results on the IP Header implementations.

```
Running test ip-header for nail...
  Testing good test cases...
    IPV4... OK
    IPV4_tcp... OK
    IPV4_udp... OK
  Testing bad test cases...
    empty... NOK
    flags_wrong-reserved... NOK
    ihl_under_5... NOK
    IPV6... NOK
    tos_wrong-reserved... NOK
    ttl_null... NOK
Test ip-header run for nail: OK
```

Sample	Category	hammer	nail	netzob	nom	parsifal	recordflux
IPV4	good	good	good	good	good	good	good
IPV4_tcp	good	good	good	good	good	good	good
IPV4_udp	good	good	good	good	good	good	good
empty	bad	bad	bad	bad	bad	bad	bad
flags_wrong-reserved	bad	bad	bad	bad	bad	bad	bad
ihl_under_5	bad	bad	bad	good	good	bad	bad
IPV6	bad	bad	bad	bad	bad	bad	bad
longer	bad	bad	bad	bad	bad	bad	bad
shorter	bad	bad	bad	bad	bad	bad	bad
tos_wrong-reserved	bad	bad	bad	bad	bad	bad	bad
ttl_null	bad	bad	bad	good	good	bad	bad

Figure 2: Execution of Nail implementation for the IP Header (on the left). Results for various IP Header parsers against the samples (on the right).

3 First results

An interesting benchmark for these tools was the DNS message format, since it allowed us to assess the expressiveness of the different tools on several aspects.

Among the tools we integrated to the platform, four of them fetaures a DNS implementation: Hammer, Kaitai Struct, Nail and Parsifal.

3.1 The DNS Specification in a nutshell

Using a pseudo description language, DNS can be modeled as follows[3]:

```
dns_query = sequence {
    id: uint16
    misc_fields: uint16
    qdcount: uin16
    ancount: uin16
    nscount: uin16
    arcount: uin16
    questions: array[qdcount] of question
    answers: array[ancount+nscount+arcount] of rr
}

question = sequence {
        qname : domain
        qtype : uint16
        qclass : uint16
}
```

```
domain = choice {
    | domain_label : sequence {
        length : uint8 = 1..64
        label : string(length)
        subdomain : domain
      }
    | domain_end : uint8 = 0
    | domain_ptr : sequence {
        ptr_magic : uint2 = 3
        ptr_offset : uint14
      }
}
```

A domain is a list of `domain_label`s (represented by variable length strings), ending either with a `domain_end` marker (which is encoded as a zero-length string), or with a `domain_ptr` pointer to an already parsed domain. The latter case is signaled by setting the two most significant bits of the first byte of a label; it allows for a simple form of compression within the message.

3.2 Expressiveness

One of the advantage of using a parsing generator is to implement complex formats with few lines of codes, which are usually mostly descriptive parts. We tried to compare the four DNS implementations at our disposal to assess the expressiveness of the corresponding tools. We first assessed the depth of the implementation (in terms of feature), then counted the number of lines each implementation required.

To compare the five implementations, we used the following samples:

- simple valid DNS requests;
- valid DNS requests with an EDNS0 extension;
- simple valid DNS responses;
- truncated DNS messages;
- messages with invalid pointers in domain names[4].

Hammer implementation only accepts simple valid DNS requests, and does not implement compression at all: pointers are simply ignored, which leads to rejecting well-formed messages.

Kaitai Struct DNS implementation correctly parses the domain names (recognizing a compression pointer), but does not interpret nor validates where the pointer points to.

Nail, which relies on a DSL and a compiler producing C code, uses the concept of *transformation*s to explicitly call a C function, external to the DSL, and apply an arbitrary computation to the data. This obviously solves the problem, but since the processing is done using a fully-fledged programming language (and not a beautiful and restricted description), there is much less guarantees on the parser correctness.

[3] The model does not describe resources records (the `rr` type) and merges the three kinds of answer a DNS message can contain to keep the explanation shorter.

[4] The samples contained

- forward pointers (including ones pointing outside the message), invalid backward pointers (with an offset not corresponding to a valid label) as well as backward pointers leading to loops.

Parsifal embeds a DSL to describe the parsers within OCaml files and requires actual OCaml code to handle DNS compression. Contrary to the Nail approach, the demarcation between descriptive parts and fully-fledged code is less obvious, since both parts live in the same file.

To count the lines, we tried to separate the descriptive parts from the "real" code parts. For Kaitai Struct, Parsifal and Nail, which rely on a DSL, the distinction is rather easy to make. For Hammer, we counted the parts using macros/helper functions corresponding to the parser combinators as description, and the rest of the files as code. The results are given in Table 3.

	Descr.	**Code**	**Total**	**Comments**
Hammer	105	158	263	The code consists in the `validate` and `act` functions.
Kaitai Struct	231	0	231	Compression pointers are not validated
Nail	39	70	109	The description and the code live in separate files.
Parsifal	130	79	209	The descriptive part is important since RRs are fully interpreted, contrary to other implementations.

Table 3: LoC counts for the different DNS implementations.

Obviously, these figures should be taken with a grain of salt, but they offer a first data point, which we would like to enrich with our platform. For example, it would be interesting to compare equivalent implementations (in terms of the files they accept/reject) to get a fairer and more precise comparison. However, from this example, it seems that Hammer is more verbose than the other tools, even though it implements a smaller subset of the format.

3.3 Robustness

At first, the four implementations did not handle correctly all our samples.

First, as already discussed, two implementations do not really handle (Hammer) or validate (Kaitai Struct) compression pointers.

For Nail, we found two bugs in the implementation, that were reported with a possible fix[5]:

- offsets larger than 256 were misinterpreted due to an operator precedence bug[6];
- the transformation function (`dnscompress_parse`) did reject forward pointers correctly, but did not check anything for backward checks, which could lead to loops in DNS compression pointers.

Finally, despite being able to properly handle all the cases, Parsifal implementation was rather lax by default, and tolerated invalid pointer (that were kept using their raw form). This did not lead to crashes or loops as with Nail, but this was not the expected behavior.

To go beyond our samples, we also tried to leverage our platform to automatically find bugs in the generated parsers. To this aim, we integrated American Fuzzy Lop (AFL) to the tools written in C: Hammer and Nail. The setup consisted of an Intel Core i5 at 2.90GHz CPU running on one core for 1-hour session.

AFL was able to find the problems we had already identified in Nail. It also helped us fix the bugs properly, after finding that our first tentative patch to avoid loops was incomplete.

For Hammer, we produced several assertion failures and a segmentation fault. The latter was reported and a fix was proposed[7]. As with the Nail issues described earlier, the bug does not lie in the description part (which consists in C macros), but in the "code" part, more specifically in the `act_domain` function, which interprets domain names once their raw representation has been parsed. The specific mistake here was a typo in the index used in a loop (`i` instead of `j`).

It would also be useful to extend the use of AFL to other tools, or to use other fuzzing frameworks (e.g. `libfuzzer`). Another idea would be to use one tool to generate samples and then test the sample against the other tools. This would help us investigate bugs related to a difference of interpretation between implementations.

[5] `https://github.com/jbangert/nail/pull/10`

[6] Actually, the bug had already been independently reported 3 years before.

[7] `https://github.com/UpstandingHackers/hammer/pull/199`

3.4 The need for a robust chain

From the design point of view, we find the external DSL approach the cleanest, since it forces the developer to explicitly signal when he steps outside the descriptive parts of his implementation. This is why we really like the DSL proposed in the Nail tool, which allows to write short and expressive descriptions, with the ability to call arbitrary C functions to handle complex transformations.

However, while implementing very simple specifications, we stumbled upon a strange bug in Nail. The minimal working example consists in the parser for a list of integers:

```
target = {
  x many uint32
}
```

Once compiled and applied to an empty file, the parser will loop for a very long time, because of an integer underflow in a generated function[8].

Beyond our surprise that such a simple specification (and sample) could trigger an unpleasant situation, we tried to fix it and to better understand the code generated by the Nail to C compiler. To this aim, we activated several compiler warnings on the generated C code, which produced a lot of output, some of which were expected for generated code (unused labels, variables or functions). But we also obtained many warnings regarding possibly dangerous integer conversions and comparisons.

Despite being fond of the Nail grammar, we find this to be extremely dangerous, because generating fragile and buggy code will eventually lead to real (and hard-to-fix) bugs, even if the original code came from a pretty DSL. Thus, we think a good parser generator should not only offer guarantees and expressiveness at its source, but also all the way down the build chain.

4 Conclusion and perspectives

In this article, we have presented a platform that is currently being developed to host different parser generators and compare them with regards to several properties: expressiveness, robustness, efficiency. At the time of writing, we support 7 different tools representing some variety in terms of programming languages and design approaches. This framework already helped us domesticate the selected tools and find some bugs or interesting behaviors.

In the short term, we would like to continue adding more tools, more specifications, as well as to develop the missing features we envision, such as performance evaluation and robustness assessment using modern fuzzing tools. In light of this goal, beyond the first contacts we have established, we plan to develop collaborations with tool maintainers to help us better implement specifications and improve the quality of our suggestions and bug reports.

We also have longer-term plans for this platform. First, comparing existing tools and DSLs might help us design a new, better description language, hopefully building on the best aspects found in the different tools. It might also give us insights (and concrete arguments) into which constructions are inherently complex and should be avoided in file formats and network protocols.

Acknowledgments

This work was supported in part by the French ANR GASP project (ANR-19-CE39-0001). The authors would also like to thank Mathias Payer for his very constructive comments and remarks.

[8] The bug report online (`https://github.com/jbangert/nail/issues/9`) gives more details about the problem.

References

[Bos14] Georges Bossert. *Exploiting Semantic for the Automatic Reverse Engineering of Communication Protocols*. PhD thesis, MATISSE, 2014.

[BZ14] Julian Bangert and Nickolai Zeldovich. Nail: A Practical Interface Generator for Data Formats. In *35. IEEE Security and Privacy Workshops, SPW 2014, San Jose, CA, USA*, pages 158–166, May 2014.

[Cou15] Geoffroy Couprie. Nom, A byte oriented, streaming, zero copy, parser combinators library in rust. In *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*, pages 142–148. IEEE Computer Society, 2015.

[Lev14] Olivier Levillain. Parsifal: A Pragmatic Solution to the Binary Parsing Problem. In *35. IEEE Security and Privacy Workshops, SPW (LangSec) 2014, San Jose, CA, USA*, pages 191–197, May 2014.

[Moc87] P.V. Mockapetris. Domain names - implementation and specification. RFC 1035 (Internet Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2673, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604, 7766, 8482, 8490, 8767.

[Pos81a] J. Postel. Internet Control Message Protocol. RFC 792 (Internet Standard), September 1981. Updated by RFCs 950, 4884, 6633, 6918.

[Pos81b] J. Postel. Internet Protocol. RFC 791 (Internet Standard), September 1981. Updated by RFCs 1349, 2474, 6864.

[PPSP06] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry L. Peterson. `binpac`: A `yacc` for Writing Application Protocol Parsers. In *Proceedings of the 6th ACM SIGCOMM Internet Measurement Conference, IMC 2006, Rio de Janeriro, Brazil*, pages 289–300, October 2006.

[RSCS19] Tobias Reiher, Alexander Senier, Jerónimo Castrillón, and Thorsten Strufe. Recordflux: Formal message specification and generation of verifiable binary parsers. In Farhad Arbab and Sung-Shik Jongmans, editors, *Formal Aspects of Component Software - 16th International Conference, FACS 2019, Amsterdam, The Netherlands, October 23-25, 2019, Proceedings*, volume 12018 of *Lecture Notes in Computer Science*, pages 170–190. Springer, 2019.

[SAH16] Robin Sommer, Johanna Amann, and Seth Hall. Spicy: a unified deep packet inspection framework for safely dissecting all your data. In Stephen Schwab, William K. Robertson, and Davide Balzarotti, editors, *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, pages 558–569. ACM, 2016.