Nail: A practical interface generator for binary formats

-How to avoid hitting your security assumptions when dealing with untrusted input-

Julian Bangert, Nickolai Zeldovich, Frans Kaashoek

Abstract

We present *Nail*, an interface generator that allows programmers to safely parse and generate protocols defined by a Parser-Expression based grammar. Nail uses a richer set of parser combinators that induce an internal representation, obviating the need to write semantic actions. Nail also provides solutions parsing common patterns such as length and offset fields within binary formats that are hard to process with existing parser generators.

1 Background

1.1 Problems handling untrusted input

Code handling untrusted user inputs is most prone to contain security vulnerabilities. While memory corruption errors or fatal logic flaws can occur anywhere in a program, bugs in the input handling code are most likely to be exploitable, because an attacker has total control over the inputs to that code.

Common wisdom thus says to take special care when preparing code that deals with untrusted input. However, with currently prevalent software design methods, it is hard to isolate input-handling code from the rest of the program, as user input is typically passed through the program and processed in bits and pieces as a 'shotgun parser'[5].

A look at the CVE database or Hacker journal such as *Phrack* shows that such design patterns and the security problems they cause are rampant. Input processing libraries such as libpng [1] or Adobe's PDF and Flash viewers are notoriously plagued with input bugs. Even simpler formats such as the *zlib* compression library have historically contained multiple security flaws[2].

Besides memory corruption vulnerabilities - against which various mitigations such as static analysis, dynamic instrumentation and various memory protection mechanisms have been developed - different implementations of the same format can also differ in their understanding of some edge cases. The security of many systems relies on different components interpreting the same data similarly.

Such parser differential vulnerabilities were found in various cryptographic systems, including iOS [3][8] and Android[4] [11] code signing and even the X.509 protocol underlying SSL [12]. Such vulnerabilities occur even if the language, runtime environment or static analysis guarantee absolute memory safety.

Finally, many protocol implementations are so complicated that even without bugs, they inadvertently provide a full Turing-complete execution environment. Examples include X86 page tables [1], ELF symbols and relocations[17]. The more powerful an input protocol is, the more control an attacker gains over a systems state - and power to manipulate a systems state is also a degree of undeserved trust. In the offensive research community, this has been generalised into treating a program as a *weird machine*[4] that operates on an input, analogous to a virtual machine operating on bytecode.

Proper input recognition has been shown to be an excellent way of eliminating malicious inputs. In one case, a hand-written PDF parser could eliminate 98% of known malicious PDFs[2].

[1] `http://www.cvedetails.com/vulnerability-list/vendor_id-7294/Libpng.html` shows 24 remote exploits in the between 2007 and 2013.

[2] `http://www.cvedetails.com/vulnerability-list/vendor_id-72/product_id-1820/GNU-Zlib.html`

[3] The XNU kernel and the user code-signing verifier interpret executable metadata differently, so the code signature sees different bytes at a virtual address than the executable that runs.

[4] Android applications are distributed as .zip files. Signatures are verified with a Java program, but the program is extracted with a C program. The java program interprets all fields as signed, whereas the C program treats them as unsigned, allowing one to replace files in a signed archive.

1.2 Parser generators

Luckily, we can avoid all of these problems at once if we have a single, executable specification of what exactly a programs input is supposed to look like. In fact, definitive grammars have long been written for all sorts of protocols and are the standard way of describing text-based protocols. Parser generators can automatically create parsers for such languages, and with common classes of grammars such as regular expressions or deterministic context free languages, that parser can be expressed in a computationally less powerful form, e.g. a finite automaton or a push down automaton, limiting the weird machine problem and enabling very efficient implementation. [13]

Specifications of binary protocols, such as RFCs, also contain grammars, although they are frequently translated into prose and diagrams as opposed to the traditional Backus-Naur-Form.

Unfortunately, grammars are seldom used directly to recognise machine-created input. For example, the security-critical and very well-engineered MIT Kerberos distribution uses parser generators, but only for handling configuration files. A notable exception is the Mongrel web server [5] which uses a grammar for HTTP written in the Ragel [18] regular expression language. Mongrel was re-written from scratch multiple times to achieve better scaleability and design, yet the grammar could be re-used across all iterations[16].

Parser generators for binary protocols were first introduced by the Hammer[6] parser. While previous parser generators could also be used to write grammars for binary protocols[7], doing so is impractically inconvenient. Hammer allows the programmer to specify a grammar in terms of bits and bytes instead of characters. Common concerns, such as endianness and bit-packing are transparently hidden by the library. Hammer implements grammars as language-integrated parser combinators, an approach popularised by Parsec for Haskell[14] and. The parser combinator style (to our knowledge, first described in [6]) is a natural way of concisely expressing top-down grammars[7] [8] by composing them from one or multiple sub-parsers. Hammer then constructs a tree of function pointers which can be invoked to parse a given input into an abstract syntax tree (AST).

Working with generic tree structures is tiresome, so the programmer can bind a *semantic action* to a sub-parser. Whenever the sub-parser is successfully invoked, the parser executes the semantic action on the output of the sub-parser and places the result as a leaf node in the syntax tree. Other parser generators frequently skip the AST entirely, requiring the programmer to specify a semantic action with every rule in the grammar.

Semantic actions are one source of vulnerability within automatically generated parsers, as they are typically implemented in C or another general purpose programming language and perform dangerous operations such as memory allocation and copying. If there is a bug in the semantic actions, the attacker is likely able to craft an input that invokes the semantic actions with controllable inputs, as the inputs have not been fully verified yet.

1.3 Motivation

Motivation While it is possible to express short transformations on the input entirely as semantic actions [9], more complicated programs usually construct an internal representation, which contains all relevant information from the input in a format native to the programming language used. For example, a C programmer ideally wants to deal with structs and NULL-terminated arrays, whereas a C++ programmer might expect STL containers, a Java programmer interfaces, a Haskell programmer records and a LISP programmer property lists. The structure of this internal representation usually resembles the structure of the grammar.

Therefore, a typical programmer has to describe the structure of his input twice or even thrice to use a parser generator. Once, to write a grammar, another time to write the semantic actions constructing his favourite intermediate representation. In most languages the programmer also has to write explicit type definitions, which describe the intermediate object model again. Most programmers will baulk at

[5] `http://mongrel2.org/`

[6] `http://github.com/upstandinghackers/hammer/`

[7] Theoretically speaking, the alphabet over which a grammar is an abstract set, so most algorithms work just as well on an alphabet of $\{0, 1\}$

[8] For more background on the history of expressing grammars, see Bryan Ford's masters thesis[9], which also describes the default parsing algorithm used by Hammer

[9] This is in fact the design rationale, to perform computation on the fly as the parser walks the parse tree.

this multiplication of efforts and either hand-write a top-down parser which still eliminates writing the semantic actions at the slight cost of verbosity or resort to passing user input through their control flow, hopefully remembering to verify each part before it is accessed.

A similar problem occurs when the programmer wants to generate output. Even though not all programs may use the same format for output as they do on input, different programs might use the same format for input and output and re-using the same grammar is a good way to save engineering effort and reduce parser differentials. Some parser generators, e.g. Boost.Spirit [10], allow the same grammar to be re-used for generating output from the intermediate representation. However, those generators require a new set of semantic actions to be written, even though the relationship between grammar and intermediate model was already expressed in the actions for the parser.

2 Design

Overview To alleviate these problems, Nail provides a richer set of combinators that induce the structure of an internal model, while also describing the grammar of the external protocol. Nail takes the combinators and produces type declarations for an internal model, the *parser*, a function to parse a sequence of bytes into an instance of the model and the *generator*, a function to create a sequence of bytes from an instance of the model.

A central design decision of Nail is that there is a bijection between the model exposed to the programmer and the bytes input and output up to syntactic equivalence. More precisely, the parser is the generators inverse, so parsing the generators output will yield the generators input, but generating the parsers output does not necessarily yield the parsers input. To understand why this makes sense, consider a grammar for a text language that tolerates white space or a binary protocol that tolerates arbitrarily long padding[11]. However, if some features of Nail are avoided, there is a proper isomorphism between model and protocol.

In the following, we present the combinators in Nail grouped by the combinators, if possible grouped by the combinator of conventional Parser Expression Grammars they replace.

Fundamental parsers The elementary parsers of Nail are the same as those of Hammer, signed and unsigned integers with arbitrary lengths up to 64 bits. Note that is possible to define parsers for sub-byte lengths, e.g. to parse the 4-bit data offset within the TCP header. Integer parsers return their value in the smallest appropriately sized machine integer type, e.g. a 24 bit integer is stored in a 32-bit wide variable.
Examples: `uint4`, `int32`

Integer parsers can be constrained to fall either within an (inclusive) range of values or be an element of a set of allowed values.
Examples: `uint8 | 1..3`,`uint16| ..512`, `int32| [1,255,512]`

Constant parser Nail also has constant parsers, which do not return a value. The simplest constant parser is an integer with fixed value, e.g. `uint8=0`. Constant parsers succeed if the inner integer parser returns the given value and fail otherwise.

Sequence: Structure Nails fundamental concept is the structure combinator. It contains a list of named parsers and unnamed constant parsers. When parsing, it parses each parser in sequence and returns a structure containing the result of each parser.

[10] `http://www.boost.org/doc/libs/1_55_0/libs/spirit/doc/html/index.html`

[11] Say, the physical layer of most communication protocols is a possibly infinite sequence of symbols that are syntactically nil followed by a pre-determined synchronisation sequence and the actual contents of the transmission.

Nail

```
header = {
 id uint16
 qr uint1
 opcode uint4
 aa uint1
 tc uint1
 rd uint1
 ra uint1
 uint3 = 0
 rcode uint4
}
```

Data Model

```
struct header{
    uint16_t id;
    uint8_t qr;
    uint8_t opcode;
    uint8_t aa;
    uint8_t tc;
    uint8_t rd;
    uint8_t ra;
    uint8_t rcode;
};
```

Repetition: Many,SepBy The many combinator takes a parser and applies it repeatedly until it fails, returning an array of the inner parsers results. SepBy additionally takes a constant parser, which it applies in between parsing two values,but not before parsing the first value or after parsing the last. Examples `many uint8`, `sepBy uint8=',' (many uint8| '0'..'9')` (which recognises comma-separated lists of numbers).

The many combinator also works on constant parsers, although as described above it can make the generator not bijective. [12] Example: `many uint8= "test"`

Semantic choice: choose We extend PEG's ordered choice combinator, which picks the first choice that succeeds, with a tag for each choice. The result is saved in a tagged union.

```
choose {
   A = uint8 | 1.. 8
   B = uint16 | ..256
}
```

Wrap The wrap combinator parses one or more constant parser, then an inner parser whose value it returns and then again parses one or more constant parsers.
Example: `<uint8='"'; many8 uint8|'a'..'z'; uint8='"'  >`

2.1 Offset fields

Another problem for parser generators is that binary protocols often contain length and offset fields. While conventional parsing algorithms can deal with bounded offset fields - a finite automaton can count a bounded integer, and we can feed the (finite) input multiple times to the finite automaton. However, this implementation is both time-inefficient - why feed many bytes into the automaton that will just be skipped - and very cumbersome to express with current parser generators. Therefore, if languages with offset fields are parsed with parser generators, the only currently feasible way is to add 'ad-hoc' hacks such as changing the input pointer of the generated parser on the fly from semantic actions.

Nail will properly support both offset and length fields and much of the following discussion applies to both , although the current prototype only implements lengths, which we will focus on.

Dependency fields We call length or offset fields *dependency fields*, because during parsing, another parser depends on them, and while generating output, their value depends on some other structure in the data model. Dependency fields appear in a structure combinator as would any other field, but their name begins with an '@' sign and they have to return an integer value. Dependency fields are not directly represented in the internal model.

Length fields: n_of The length combinator takes a dependency field and a parser, evaluating the parser as often as the value of the dependency field, returning an array of the values. When generating output, it emits the array and writes its length to the dependency field.

[12] By default, the generator will emit exactly one iteration of the constant parser in its output.

2.2 Example Grammars

To further familiarise the reader with Nail, we provide two example grammars.

UTF-16 This grammar recognises valid UTF-16 strings and exposes an array of code points.

Nail

```
utfstring = many choose {
          SUPP= {
                lead uint16 | 55296..56319
                            // 0xD800..0xDBFF
                trail uint16 | 56320..57343
                            // 0xDC00..0xDFFF
                }
          BASIC = uint16 | !55296..57343
        }
```

Data Model

```
struct utfstring {
 struct {
   enum  {SUPP,BASIC} N_type;
   union {
    struct {
     uint16_t lead;
     uint16_t trail;
    } SUPP;
    uint16_t BASIC;
   };
 }*elem;
 size_t count;
};
```

DNS packet This grammar recognises DNS packets without label compression, as per RFC1035

```
labels = <many {@length  uint8 | 1..255
                label n_of @length uint8 }
            uint8 = 0>
question= {   labels labels
              qtype uint16 | 1..16
              qclass uint16 | [1,255]
               }
answer  = {
            labels labels
            rtype uint16 | 1..16
            class uint16 | [1]
            ttl uint32
            @rlength uint16
            rdata n_of @rlength uint8
            }
dnspacket = {id uint16
              qr uint1
              opcode uint4
              aa uint1
              tc uint1
              rd uint1
              ra uint1
              uint3 = 0
              rcode uint4
              @questioncount uint16
              @answercount uint16
              authoritycount uint16   // Ignored
              additionalcount uint16  //Ignored
              questions n_of @questioncount question
              responses n_of @answercount answer
            }
```

3 Implementation

The current prototype of the Nail parser generator supports the C programming language and top down parsers. Options for C++ STL data models and emitting Packrat parser [10] are under heavy development. In this section, we will discuss some particular features of our parser implementation.

Parsing A generated Nail parser makes two passes through the input , the first to validate and recognise the input, the second to bind this data to the internal model. Currently the parser is a straightforward top down parser, although facilities have been made to add Packrat parsing to achieve linear parsing times.

Memory allocation Many security vulnerabilities can occur when heap allocations are done improperly. Therefore, just like Hammer, Nail avoids using the heap as much as possible, using a custom arena allocator and allocating only fixed size blocks from the system malloc. However, Nail extends upon Hammers approach and uses two arenas for each parsed input. One arena is used for intermediate results and is freed(and zeroed) after parsing completes, whereas the other arena is used only for allocating the result and has to be freed by the programmer.

Intermediate representation Most parser generators, such as Bison do not have to dynamically allocate temporary data, as they evaluate a semantic action on every rule. However, as our goal is to perform as little computation as possible without having validated the input, and we do not want to mix temporary objects with the results of our parse, we need some intermediate structure to cache parse results. This structure is also used as a value in the Packrat hash table.

Hammer solves this problem by storing a full abstract syntax tree. However, this abstract syntax tree is at least an order of magnitude larger than the input, because it stores a large tree node structure for each input byte and for each rule reduced. This allows Hammer semantic actions to get all necessary information without ever seeing the raw input stream. However, because we also automatically generate our second pass, which corresponds to Hammers semantic actions, we can trust it as much as we trust the parser and thus expose it to the raw input stream.

Under this premise, the actions need very limited information from the recogniser to correctly handle the input stream. In particular, the parsers control flow only branches at the choice and repeat combinators. For all other combinators, the action just has to assign a fixed sequence of fundamental parsers to a fixed sequence of internal model fields. Thus we can represent the parsers success as a sequence of selected choices and counted repeats and use this to reconstruct the syntax of the input.

Dependency Fields During parsing, dependency fields occur before the context in which they are used. The parser stores their value and retrieves it afterwards when encountering the combinator that uses them. When generating output, the dependency field is first filled with a filler value, then later when the first combinator that determines this fields value is encountered, the field is overwritten. Any further combinators using this dependency will then validate that the dependency field is correct.

Bootstrapping To demonstrate the feasibility of the nail parser generator, our parser generator uses a Nail parser to recognise Nail grammars. A superset of the grammar language described in this paper is implemented in 100 lines of Nail, which feed into about a thousand lines of C++ that implement the actual parser generator. Bootstrapping is supported via an LALR implementation.

4 Evaluation

In order to test the Nail framework, we cloned the test DNS server from the Hammer distribution to Nail. Hammer ships with a toy DNS server written in 683 lines of code, excluding the hammer framework itself that responds to any valid DNS query with a CNAME record to the domain “spargelze.it”. Most of this code is taken up with custom validators, semantic actions and data structure definitions.

Our DNS server measures 148 lines of C and 48 lines of Nail grammar, and supports a custom zone file format with A,NS,MX and CNAME records. The same grammar is used together with 98 lines of C to implement a functional toy clone of the host command line tool. However, because our grammar

does not yet support DNS label compression, the latter tool will occasionally reject valid real world DNS responses. Both clients have functional anti-spoofing measures.

It is hard to compare our efforts to a real world DNS server, as we have less functionality, in particular for DNS compression and additional hint records than real world DNS servers. However, the closest in functionality and intent is Dan Bernstein's djbdns[13], which aims to be a minimalist, highly secure DNS server. The latest release of DJBDNS, however, including various support tools is about ten thousand lines of C as measured by SLOCCOUNT. We are confident that it is possible to a feature-par version with Nail that is an order of magnitude smaller and intend to do so.

5 Future Work

The Nail parser generator is work in progress and feedback of all sorts is heavily encouraged. Source code is available upon request and will be released publicly pending clean-up and maturity. Short-term future work will include improving the scoping of dependency fields and adding support for offset fields.

A major problem with the current design of Nail is that the design of the grammar dictates the interna structure of the software. This makes changing grammars or adding Nail to existing software awkward. One possible solution to this problem would be to implement a concept similar to Relational Lenses [3], which would allow the data model to be 'seen' by the rest of the program through an isomorphism. Such an isomorphism would still be much more concise than two sets of semantic actions, while allowing changes in syntax, alternative representations and adaption to legacy systems.

Finally, we would like to demonstrate the capabilities of Nail by implementing various binary formats 'notorious' for their insecurity in Nail. Nail was designed with the idioms of formats such as PDF and PNG in mind. Ultimately, we want to provide examples of successful Nail parsers all throughout a network stack - from a user-space TCP stack to a PNG de-compressor.

5.1 Previous work

Generating parsers and generators from an executable specification is the core concept of Interface Generators, e.g. in CORBA or more recently[19]. However, while interface generators work very well for existing grammars, they do not allow full control over the format of the output, so cannot be used to implement legacy protocols. Very related work has been done at Bell Labs with the PacketTypes system[15], however PacketTypes works only as a parser and does not support the expressive power of PEGs, but rather implements a C like structure model enhanced with data-dependent length fields and constraints.

References

[1] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W Smith. The page-fault weird machine: lessons in instruction-less computation. In *Presented as part of the 7th USENIX Workshop on Offensive Technologies*. USENIX, 2013.

[2] Andreas Bogk. Certified programming with dependent types.

[3] Aaron Bohannon, Benjamin C Pierce, and Jeffrey A Vaughan. Relational lenses: a language for updatable views. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 338–347. ACM, 2006.

[4] Sergey Bratus, Michael E Locasto, Meredith L Patterson, Len Sassaman, and Anna Shubina. Exploit programming: From buffer overflows to weird machines and theory of computation. *USENIX; login*, pages 13–21, 2011.

[5] Sergey Bratus, Meredith L. Patteron, and Dan Hirsch. From "shotgun parsers" to more secure stacks. In *Shmoocon*, 2013.

[6] William H Burge. *Recursive programming techniques*. Addison-Wesley Reading, 1975.

[7] Nils Anders Danielsson. Total Parser Combinators. In *Proc. 15th ACM SIGPLAN Int'l Conf. on Functional Programming*, ICFP '10, pages 285–296, 2010.

[13] `http://cr.yp.to/djbdns.html`

[8] Team Evaders. Swiping through modern security features. HITB Amsterdam, 2013, 2013.

[9] Bryan Ford. *Packrat parsing: a practical linear-time algorithm with backtracking*. PhD thesis, Massachusetts Institute of Technology, 2002.

[10] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *Proceedings of the 2002 International Conference on Functional Programming*, Oct 2002.

[11] Jay Freeman. Yet another android master key bug. Technical report, Saurik, 2013. `http://www.saurik.com/id/19`.

[12] Dan Kaminsky, Meredith L. Patterson, and Len Sassaman. PKI Layer Cake: New Collision Attacks against the Global X.509 Infrastructure. In *Financial Cryptography*, pages 289–303. Springer, 2010.

[13] Donald E. Knuth. On the Translation of Languages from Left to Right. *Information and Control*, 8(6):607 – 639, 1965.

[14] Daan Leijen and Erik Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[15] Peter J McCann and Satish Chandra. Packet types: abstract specification of network protocol messages. In *ACM SIGCOMM Computer Communication Review*, volume 30, pages 321–333. ACM, 2000.

[16] Meredith Patterson. Langsec 2011-2016. `http://prezi.com/rhlij_momvrx/langsec-2011-2016/`.

[17] Rebecca Shapiro, Sergey Bratus, and Sean W Smith. âĂIJweird machinesâĂİ in elf: A spotlight on the underappreciated metadata. In *Presented as part of the 7th USENIX Workshop on Offensive Technologies*. USENIX, 2013.

[18] Adrian D. Thurston. Parsing computer languages with an automaton compiled from a single regular expression. In *Proceedings of the 11th International Conference on Implementation and Application of Automata*, CIAA'06, pages 285–286, Berlin, Heidelberg, 2006. Springer-Verlag.

[19] Kenton Varda. Protocol buffers: Google's data interchange format. Technical report, Google, 6 2008.