Nail: A practical interface generator for data formats

Julian Bangert and Nickolai Zeldovich
MIT CSAIL

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 Introduction

Code that handles untrusted inputs, such as processing network data or parsing a file, is error-prone and is often exploited by attackers. This is in part because attackers have precise control over the inputs to that code, and can craft inputs that trigger subtle corner cases in input processing. For example, the libpng library has had 24 remotely exploitable vulnerabilities from 2007 to 2013,[1] and Adobe's PDF and Flash viewers have been notoriously plagued by input processing vulnerabilities. Even relatively simple formats, such as those used by the zlib compression library, have had input processing vulnerabilities in the past.[2]

A promising approach to avoid such vulnerabilities is to specify a precise grammar for the input data format, and to use a parser generator, such as `lex` and `yacc`, to synthesize the input processing code. Developers that use a parser generator do not need to write error-prone input processing code on their own, and as long as the parser generator is bug-free, the application will be safe from input processing vulnerabilities. Unfortunately, applying this approach in practice, using state-of-the-art parser generators, still requires too much manual programmer effort, making it error-prone, as we describe next.

First, parser generators typically parse inputs into an abstract syntax tree (AST) that corresponds to the grammar. In order to produce a data structure that the rest of the application code can easily process, application developers must write explicit *semantic actions* that update the application's internal representation of the data based on each AST node. Writing these semantic actions by hand is error-prone, much like other input processing code, and mistakes can result in memory corruption bugs or misinterpreted inputs. Writing these semantic actions also requires the programmer to describe the structure of the input three times—once to describe the grammar, once to describe the internal data structure, and once again in the semantic actions that translate the grammar into the data structure—leading to another potential source of bugs and inconsistencies.

Second, applications often need to produce output in the same format as their input—for example, applications might both read and write files, or both receive and send network packets. Most parser generators just focus on parsing an input, rather than producing an output, thus requiring the programmer to manually construct outputs, which is error-prone. Some parser generators, such as Boost.Spirit [7], allow reusing the grammar for generating output from the internal representation. However, those generators require yet another set of semantic actions to be written, transforming the internal representation into an AST.

Third, data formats, especially binary formats such as PNG or PDF, can have *structural dependencies*, such as offset, length, and checksum fields that are hard to express in state-of-the-art grammar languages. Often the programmer is required to manually write control code, such as re-positioning the parser's input stream, looping over the invocation of a sub-parser, or computing a checksum over raw input bytes. Besides leaving much room for errors with offset arithmetic, such code is usually not reusable when generating output.

This paper presents the initial design and implementation of Nail, a parser generator that greatly reduces the programmer effort required to use a grammar-based parser. Nail addresses the above three challenges with several key ideas.

First, Nail reduces the expressiveness of its grammar language by removing semantic actions. Existing parser generators allow arbitrary computation to transform between the AST and the parser output. Instead, Nail derives the structure of its output automatically from the grammar, forcing the programmer to clearly separate

[1]`http://www.cvedetails.com/vulnerability-list/vendor_id-7294/Libpng.html`
[2]`http://www.cvedetails.com/vulnerability-list/vendor_id-72/product_id-1820/GNU-Zlib.html`

syntactic validation and semantic processing.

Second, this well-defined internal representation allows Nail to invert the parser and generate output. However, Nail allows *constants* in the external format to have multiple representations if they should not affect the semantics of the data. For example, in a text protocol, the amount of white-space separating tokens should not affect the meaning of the data and consequently Nail does not expose it to the application. As long as constants are only used for their intended purpose of representing syntax-only features, the generated output will have the same semantics as the parsed input.

Third, Nail provides support for structural dependencies, transparently handling offset and length fields. By hiding the offset and length values from the programmer, Nail ensures they remain consistent even if the data is changed by the application.

We have implemented an initial prototype of Nail for C. Our experience so far suggests that it is a promising approach: we were able to construct a succinct grammar for DNS packets, and write a small DNS server that uses Nail for all packet input and output, and operates purely on Nail-generated data structures, with no need for any semantic actions.

The rest of this paper is organized as follows. §2 puts Nail in the context of related work. §3 describes Nail's design. §4 discusses our initial implementation of Nail. §5 provides some initial evaluation results. §6 suggests several directions for future work, and §7 concludes.

2 Related work

Language safety. Input processing vulnerabilities fall into two broad classes. The first class is memory safety bugs, such as buffer overflows, which allow an adversary to corrupt the application's memory using specially crafted inputs. These mistakes arise in lower-level languages that do not provide memory safety guarantees (such as C), and can be partially mitigated by a wide range of techniques, such as static analysis, dynamic instrumentation, and address space layout randomization, that make it more difficult for an adversary to exploit these bugs. Nail helps developers of lower-level languages avoid these bugs in the first place.

The second class is logic errors, where application code misinterprets input data. This can lead to serious security consequences when two systems disagree on the meaning of a network packet or a signed message, as in iOS[3] [8] and Android[4] [11] code signing and even the X.509 protocol underlying SSL [12]. These mistakes are highly application-specific, and are difficult to mitigate using existing techniques, and these mistakes can occur even in high-level languages that guarantee memory safety. By allowing developers to specify their data format just once, Nail avoids logic errors and inconsistencies in parsing and output generation.

A subclass of logic errors are so-called *weird machines*, where implementation side effects or underspecified parser behavior leads to a protocol or data format inadvertently becoming a Turing-complete execution environment, even though the original grammar did not require it. Frequently, this execution environment can either then directly manipulate data in unwanted ways or be used to make exploiting another bug feasible.[5] Examples include x86 page tables [1], and ELF symbols and relocations [17]. In the offensive research community, this has been generalized into treating a program as a *weird machine* [4] that operates on an input, analogous to a virtual machine operating on bytecode. Nail avoids these problems by having the parser precisely match the specified grammar, eliminating under-specified behavior.

Parsing frameworks. Proper input recognition has been shown to be an excellent way of eliminating malicious inputs. In one case, a PDF parser implemented in Coq could eliminate over 95% of known malicious PDFs [2]. However, manually writing parser code does not scale to the number of file formats and protocols in existence and might result in parser code tied to one specific application.

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 [14], however PacketTypes works only as a parser, not as an output generator and does not support the expressive power of parsing expression grammars (PEGs), but rather implements a C-like structure model enhanced with data-dependent length fields and constraints.

Parser generators for binary protocols were first introduced by the Hammer [16] parser. While previous parser generators could also be used to write grammars for

[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, thereby undermining Android's security model.

[5]For example, by compiling a return-oriented-programming exploit from code fragments discovered on the fly.

binary protocols,[6] doing so is practically 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 popularized by Parsec for Haskell [13]. The parser combinator style (to our knowledge, first described in [5]) is a natural way of concisely expressing top-down grammars [6][7] 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).

Nail improves upon Hammer in three ways. First, Nail generates output besides recognizing input. Second, Nail does not require the programmer to write potentially insecure semantic actions. Last, Nail's structural dependencies allow it to work with protocols Hammer cannot recognize, such as protocols with offset fields or length fields (Hammer has extremely limited support for length fields: it can parse arrays immediately preceded by their length).

Application use of parsers. Generated parsers have long been used to parse human input, such as programming languages and configuration files. Frequently, such languages are often specified with a formal grammar in an executable form.

Unfortunately, parser frameworks are seldom used to recognize machine-created input. For example, the security-critical and well-engineered MIT Kerberos distribution uses parser generators, but only for handling configuration files. A notable exception is the Mongrel web server[8] 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 scalability and design, yet the grammar could be re-used across all iterations [15].

3 Design

3.1 Overview

To integrate the data format and the internal representation, Nail provides a rich set of combinators that not only describe the grammar of the external protocol, but also induce an internal model.

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

[7]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.

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

Syntax	Semantics
`int32`	32-bit signed integer
`uint4`	4-bit unsigned integer, returned as an 8-bit value
`uint8 | 1..3`	8-bit unsigned integer, $1 \leq x \leq 3$
`uint16 | ..512`	unsigned 16-bit integer, $x \leq 512$
`int32 | [1,255,512]`	signed 32 bit integer, $x \in \{1,255,512\}$

Figure 1: Example Nail grammars for integer values.

A central design decision of Nail is that there is a *semantic bijection* between the model exposed to the programmer and the byte-level input and output, up to syntactic equivalence for unambiguous grammars. More precisely, the parser is the generator's inverse, so parsing the generator's output will yield the generator's 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.[9] In that case, the program semantics should be independent of the number of padding elements in the input, and Nail therefore does not expose that information to the programmer. We call such discarded fields *constants*. Currently, Nail always makes a default choice when there are multiple options to express a constant, however Nail could be extended to allow a grammar-specific plug-in to make these choices, say for faster alignment, streaming applications when data is not ready, or visual appearance in human-readable protocols. Similarly, Nail does not preserve the layout of objects referred to by their offsets. If the grammar contains no constants and offset fields, there is a proper isomorphism between model and protocol.

Nail evaluates the combinators and produces:

- type declarations for the 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.

In the rest of this section, we present Nail's combinators.

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

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

```
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;
};
```

Figure 2: Nail grammar (left) and data model (right) for a part of the grammar for DNS packets. The `uint3 = 0` grammar represents 3 bits of padding (filled with zeroes).

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; in Nail's syntax. 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.

Integer parsers can be constrained to fall either within an (inclusive) range of values or be an element of a set of allowed values. Figure 1 shows several examples. Invalid values or prematurely reaching the end of input will raise an error when parsing input or generating output, and abort the procedure.

Sequence. Nail's fundamental concept is the structure combinator. It contains a list of named parsers and unnamed constant parsers. The parser invokes each field in sequence and returns a structure containing the result of each parser. For example, Figure 2 shows the structure combinator from a part of the grammar for DNS packets, along with the data model corresponding to it.

Repetition. The *many* combinator takes a parser and applies it repeatedly until it fails, returning an array of the inner parsers results. The *sepBy* combinator 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. For example, `many uint8` represents an array of 8-bit unsigned integers, and `sepBy uint8=',' (many uint8 | '0'..'9')` recognizes comma-separated lists of decimal numbers.

Semantic choice. We extend a parsing expression grammar's ordered choice combinator with a tag for each choice. The parser attempts to parse each option in the order they are specified in the field and stores the result in a tagged union. If an option fails, the parser backtracks

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

Figure 3: A simple choice combinator that parses either an 8-bit unsigned integer with a value between 1 and 8 (option A), or a 16-bit unsigned integer with a value of at most 256 (option B).

to the beginning of the choice combinator's input. Care must be taken that the choices do not overlap, because Nail always picks the first successfully parsed choice. If two options overlap, generated output for the latter option is not necessarily understood identically by the parser. However, actual data formats are usually not ambiguous in this sense. Figure 3 demonstrates a simple choice combinator.

3.2 Constant parsers

Nail also features *constant parsers*, which do not affect the internal representation. Constant parsers can appear instead of structure fields and as separators in the sepBy combinator.

The simplest constant parser is an integer or array of integers with fixed value; for example, `uint8=0`, or `many uint8=[1,2]`. A convenience notation for strings is also supported: `many uint8 = "foo"`.

Wrap combinator. When implementing real protocols with Nail, we often found that structures that consist of many constant parsers and only one named field. This pattern is common in binary protocols which use fixed headers to denote the type of data structure to be parsed. In order to keep the internal representation cleaner, we introduced the wrap combinator, which takes a sequence of parsers containing exactly one non-constant parser. The parser and generator act as though the wrap combinator was a sequence of parsers, but the data model does not wrap the single value in another structure, making the application-visible representation (and thus application code) more concise.

For example, `<uint8='"'; many int8 | 'a'..'z'; uint8='"'>` parses a quoted lowercase word into an array, excluding the quotation marks.

Constant combinators. In some protocols, there might be many ways to represent the same constant field and there is no semantic difference between the different syntactic representations. To support this pattern, Nail allows developers to use choice and repetition combinators together with constant fields, such as `many` (uint8=' ') (representing any number of space characters), or || uint8 = 0x90 || uint16 = 0x1F0F (parsing two of the many representations of NOP on the x86 architecture). Note that constant may have varying lengths. This is particularly useful for handling padding or whitespace.

As discussed above, choosing to use these combinators on constant parsers removes the bijection between byte-strings and our data model, as there are multiple byte-strings that correspond to the same internal data structure and the generator has to choose one of these representations. As such, constant combinators are the generator dual of ambiguous choice combinators in the parser, because they lead to ambiguities in the generator.

3.3 Dependent fields

Another problem for parser generators is that binary protocols often contain length and offset fields. Conventional parsing algorithms can, in principle, 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 (it feeds 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 need to be parsed with parser generators, the only currently practical way is to add ad-hoc hacks such as changing the input pointer of the generated parser on the fly, as part of the code in the semantic action for the offset field.

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.

We call length or offset fields *dependent 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 integer field, but their name begins with an @ sign. A dependency field has to appear in the grammar before it can be used.

Dependency fields are not exposed in the data model, but instead are transparently computed. This frees the developer from checking that these fields are correct (for input) or having to keep their values in sync with the rest of the data structure (for output).

Length fields. The length combinator, n_of, takes a dependency field *n* and a parser, evaluates the parser exactly *n* times (i.e., setting the number of iterations to be the *n* field's value), and returns an array of the parser's values. When generating output, it emits the array and writes its length to the dependency field.

Offsets. The offset combinator takes a dependency field and a parser. It moves the parser to the position specified in the offset field and invokes the inner parser, and then moves the input back to its original position. While generating output, all structures referred to by offset are generated after the main structure and the dependent offset fields are patched up.

Checksums. In many data formats, some values depend on external representation, such as checksums and cryptographic signatures. While it would be possible to extend our constraint language to be powerful enough to support such constructs, we would essentially be building a separate, Turing-complete language that has all the same pitfalls existing programs have. Therefore, we intend to allow the programmer to carefully escape Nail's programming model and write a function that takes a pointer to the dependent value and a range of bytes, using the raw_depend combinator.

For example, we imagine the following grammar could be used to represent a sequence of bytes data followed by its CRC32 checksum:

```
data many uint8; @checksum uint32
raw_depend @checksum data crc32
```

where crc32 is a function supplied by the application, with the following signature:

```
bool crc32(uint32_t *out, uint8_t *in);
```

Because this feature compromises Nail's security guarantees, it should only be used in limited circumstances and with carefully prepared checksum functions. This feature is not implemented in the current prototype.

4 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 development. In this section, we will discuss some particular features of our parser implementation.

The source code of our implementation, together with the examples described in §5 are available on GitHub at https://github.com/jbangert/nail.

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

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

Figure 4: Nail grammar (left) and data model (right) for UTF-16 strings.

Parsing. A generated Nail parser makes two passes through the input: the first to validate and recognize the input, and 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 allocator (malloc). However, Nail extends upon Hammer's approach and uses two arenas for each parsed input. One arena is used for intermediate results and is released (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 before the input has been validated, and we do not want to mix temporary objects with the results of our parse, we use an append-only trace to store intermediate parser results.

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 of the necessary information without ever seeing the raw input stream. However, because we also automatically generate our second pass, which corresponds to Hammer's semantic actions, we can trust it as much as we trust the parser, and thus can expose it to the raw input stream.

Under this premise, the actions need limited information from the recognizer to correctly handle the input stream. In particular, the parser's control flow branches only at the choice, repetition, and constant combinators. Thus, for each of those combinators, we store the minimum amount of information required to reconstruct the syntactic structure of the input. The trace is an array of integers. Whenever the parser encounters a choice, it appends two integers to the trace. After it successfully parses a choice, the parser writes the number of that choice and the length of the trace when it began parsing that choice. When backtracking in the input, the parser does not backtrack in the trace. This means that offsets within the trace can be used for a Packrat hash-table to memoize backtrack-heavy parsers.

When encountering a repetition combinator, the parser records the number of times the inner parser succeeded, and when encountering a constant parser of variable size, it records how much input was consumed by the constant parser.

In a second pass, the parser then allocates the internal representation from an arena allocator and binds the fields to values from the input, while following the trace to determine how many array fields to parse and which choices to pick.

Dependency fields. During parsing, dependency fields occur before the context in which they are used. The parser stores their values and retrieves them 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 recognize Nail grammars. A superset of the grammar language described in this paper is implemented in 100 lines of Nail, which feed into about 1,000 lines of C++ that implement the actual parser generator. Bootstrapping is supported via an implementation of the Nail language in the Lemon parser generator, a variant of Yacc.

5 Evaluation

In our preliminary evaluation of Nail, we try to answer two questions:

- Can Nail grammars support real-world data formats?
- How much programmer effort is required to build an application that uses Nail for data input and output?

Data formats. To answer the first question, we implemented two Nail grammars: one for UTF-16 encoded strings, exposing an array of code points (shown in Figure 4), and another for a subset of DNS packets sent to and from an authoritative name server, without label compression, as per RFC1035 (shown in Figure 5). Furthermore, our GitHub repository contains other Nail grammars, such as a TAP network stack that processes Ethernet, ARP, ICMP, IP, and UDP packets, and the grammar for Nail itself.[10] The results suggest that Nail is a good fit for these data formats.

Programmer effort. To answer the second question, we implemented a functioning toy DNS server. In particular, 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, with only 52 lines of code defining the grammar with Hammer's combinators.

Our DNS server consists of 148 lines of C code 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

[10]https://github.com/jbangert/nail/tree/master/examples

```
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
  @qc uint16
  @ac uint16
  uint16 = 0    // authority
  uint16 = 0    // additional
  // We don't support authority or
  // additional sections in the prototype
  questions n_of @qc question
  responses n_of @ac answer
}
```

Figure 5: Nail grammar for DNS packets, used by our prototype DNS server.

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 the programming effort required to implement our toy DNS server to that of a real world DNS server, since we have less functionality, in particular for DNS compression and additional hint records that real-world DNS servers send. However, the closest in functionality and intent is Dan Bernstein's djbdns,[11] which aims to be a minimalist, highly secure DNS server. The latest release of djbdns, including various support tools, is about 10,000 lines of C as measured by `sloccount`. We expect that it is possible to build a feature-par version with Nail that is an order of magnitude smaller and intend to do so.

Other issues. As Nail is work in progress, many parts of the implementation, syntax and design are not complete yet and we do not yet have meaningful performance or security metrics.

6 Future work

The Nail parser generator is currently a work-in-progress, and we would appreciate feedback on our initial design and prototype implementation. Short-term next steps include improving the scoping of dependency fields and adding support for offset fields.

One problem with the current design of Nail is that the design of the grammar dictates the internal 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 throughout a network stack, from a user-space TCP stack to a PNG de-compressor.

7 Conclusion

This paper presented the initial design and implementation of *Nail*, an interface generator for data formats. Nail helps programmers to avoid memory corruption and ambiguity vulnerabilities while reducing effort in parsing and generating real-world protocols and file formats. Nail achieves this by reducing the expressive power of the grammar, maintaining a *semantic bijection* between data formats and internal representations, and allowing programmers to specify *structural dependencies* in the data format. Preliminary experience with implementing a DNS server using Nail suggests that this is a promising approach.

The source code for our prototype of Nail is available at `https://github.com/jbangert/nail`.

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

Acknowledgments

We thank M. Frans Kaashoek and the anonymous reviewers for their feedback. This research was supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, and by NSF award CNS-1053143.

References

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

[2] Andreas Bogk. Certified programming with dependent types. Chaos Communication Camp, August 2011. `http://www.youtube.com/watch?v=CmPw7eo3nQI`.

[3] Aaron Bohannon, Benjamin C Pierce, and Jeffrey A Vaughan. Relational lenses: a language for updatable views. In *Proceedings of the 25th ACM Symposium on Principles of Database Systems*, pages 338–347, Chicago, IL, June 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. *;login: The Magazine of Usenix & Sage*, 36(6):13–21, December 2011.

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

[6] Nils Anders Danielsson. Total parser combinators. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 285–296, Baltimore, MD, September 2010.

[7] Joel de Guzman and Hartmut Kaiser. Boost Spirit 2.5.2, October 2013. http://www.boost.org/doc/libs/1_55_0/libs/spirit/doc/html/.

[8] Team Evaders. Swiping through modern security features. In *Proceedings of the HITB Amsterdam*, 2013.

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

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

[11] Jay Freeman. Yet another Android master key bug, 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 *Proceedings of the 2010 Conference on Financial Cryptography and Data Security*, pages 289–303, January 2010.

[13] 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.

[14] Peter J McCann and Satish Chandra. Packet types: abstract specification of network protocol messages. *ACM SIGCOMM Computer Communication Review*, 30(4):321–333, 2000.

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

[16] Meredith Patterson and Dan Hirsch. Hammer parser generator, March 2014. https://github.com/UpstandingHackers/hammer.

[17] Rebecca Shapiro, Sergey Bratus, and Sean W Smith. "Weird machines" in ELF: A spotlight on the underappreciated metadata. In *Proceedings of the 7th USENIX Workshop on Offensive Technologies*, Washington, DC, August 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*, pages 285–286, Taipei, Taiwan, 2006.

[19] Kenton Varda. Protocol buffers: Google's data interchange format, June 2008. http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html.