Nail
A Practical Tool for Parsing and Generating Data Formats

JULIAN BANGERT AND NICKOLAI ZELDOVICH

Julian Bangert is a second year PhD student working on computer security at MIT. When he is not building parsers for complicated formats, he is interested is building exploit mitigation techniques, side-channel resistant cryptography, and finding Turing-complete weird machines in unexpected places, such as your processor's virtual memory system. julian@csail.mit.edu.

Nickolai Zeldovich is an associate professor at MIT's Department of Electrical Engineering and Computer Science and a member of the Computer Science and Artificial Intelligence Laboratory. His research interests are in building practical secure systems, from operating systems and hardware to programming languages and security analysis tools. He received his PhD from Stanford University in 2008, where he developed HiStar, an operating system designed to minimize the amount of trusted code by controlling information flow. In 2005, he co-founded MokaFive, a company focused on improving desktop management and mobility using x86 virtualization. Professor Zeldovich received a Sloan Fellowship (2010), an NSF CAREER Award (2011), the MIT EECS Spira Teaching Award (2013), and the MIT Edgerton Faculty Achievement Award (2014). nickolai@csail.mit.edu

I am ZIP, file of files. Parse me, ye mighty and drop a shell.

—*Edward Shelley on the Android Master Key*

Binary file formats and network protocols are hard to parse safely: The libpng image decompression library had 24 remotely exploitable vulnerabilities from 2007 to 2013. According to CVEdetails, Adobe's PDF and Flash viewers have been notoriously plagued by input processing vulnerabilities, and even the zlib compression library had input processing vulnerabilities in the past. Most of these attacks involve memory corruption—therefore, it is easy to assume that solving memory corruption will end all our woes when handling untrusted inputs.

However, as memory-safe languages and exploit mitigation tricks are becoming more prevalent, attackers are moving to a new class of attack—parser differentials. Many applications use handwritten input processing code, which is often mixed with the rest of the application—e.g., by passing a pointer to the raw input through the application. This (anti-) pattern makes it impossible to figure out whether two implementations of the same format or protocol are identical, and input handling code can't be easily reused between applications. As a result, different applications often disagree in the corner cases of a protocol, which can have fatal security consequences. For example, Android has two parsers for ZIP archives involved in securely installing applications. First, a Java program checks the signatures of files contained within an app archive and then another tool extracts them to the file system. Because the two ZIP parsers disagree in multiple places, attackers can modify a valid file so that the verifier will see the original contents, but attacker-controlled files will be extracted to the file system, bypassing Android's code signing. Similar issues showed up on iOS [3] and SSL [4].

Instead of attempting to parse inputs by hand and failing, a promising approach is to specify a precise grammar for the input data format and automatically generate parsing code from that with tools like yacc. As long as the parser generator is bug-free, the application will be safe from many input processing vulnerabilities. Grammars can also be reused between applications, further reducing effort and eliminating inconsistencies.

This approach is typical in compiler design and in other applications handling text-based inputs, but not common for binary inputs. The Hammer framework [5] and data description languages such as PADS [2] have been developing generated parsers for binary protocols.

However, if you wanted to use existing tools to parse PDF or ZIP, you would soon find that they cannot handle the complicated—and therefore most error-prone—aspects of such formats, so you'd still have to handwrite the riskiest bits of code. For example, existing parser generators cannot conveniently represent size or offset fields, and more complex features, such as data compression or checksums, cannot be expressed at all.

```
1  dnspacket =
2 {
3   id uint16
4   qr uint1
5   opcode uint4
6   aa uint1
7   tc uint1
8   rd uint1
9   ra uint1
10   uint3 = 0
11   rcode uint4
12   @qc uint16
13   @ac uint16
14   @ns uint16
15   @ar uint16
16   questions n_of @qc question
17   responses n_of @ac answer
18   authority n_of @ns answer
19   additional n_of @ar answer
20 }
21 question = {
22   labels compressed_labels
23   qtype uint16 | 1..16
24   qclass uint16 | [1,255]
25 }
26 answer = {
27   labels compressed_labels
28   rtype uint16 | 1..16
29   class uint16 | [1]
30   ttl uint32
31   @rlength uint16
32   rdata n_of @rlength uint8
33 }
34 compressed_labels = {
35   $decompressed transform dnscompress ($current)
36   labels apply $decompressed labels
37 }
38 label = { @length uint8 | 1..64
39           label n_of @length uint8 }
40 labels = <many label; uint8 = 0>
```

Figure 1: Nail grammar for DNS packets, used by our prototype DNS server

Furthermore, some parser generators are cumbersome to use when parsing binary data for several reasons. First, many parser generators don't produce convenient data structures, but call semantic actions that you have to write to build up a data structure your program can use. Therefore, you must describe the format up to three times—in the grammar, the data structure, and the semantic actions. Second, most parser generators only address parsing inputs, so you have to manually construct outputs. Some parser generators, such as Boost.Spirit, allow generating output but require you to write another set of semantic actions.

We address these challenges with Nail, a new parser generator for binary formats. First, Nail grammars describe not only a format, but also a data type to represent it within the program. Therefore, you don't have to write semantic actions and type declarations, and you can no longer combine syntactic validation and semantic processing. Second, Nail will also generate output from this data type without requiring you to write more risky code or giving you a chance to introduce ambiguity.

Third, Nail introduces two abstractions, *dependent fields* and *transformations*, to elegantly handle problematic structures, such as offset fields or checksums. Dependent fields capture fields in a protocol whose value depends in some way on the value or layout of other parts of the format; for example, offset or length fields, which specify the position or length of another data structure, fall into this category. Transformations allow you to write plugins, allowing your programs to handle complicated structures, while keeping Nail itself small, yet flexible.

In the rest of this article, we will show some tricky features of real-world formats and how to handle them with Nail.

Design by Example

In this section, we will explain how to handle basic data formats in Nail, how to handle redundancies in the format with dependent fields, and how Nail parsers can be extended with transformations.

As a motivating example, we will parse DNS packets, as defined in RFC 1035. Each DNS packet consists of a header, a set of question records, and a set of answer records. Domain names in both queries and answers are encoded as a sequence of labels, terminated by a zero byte. Labels are Pascal-style strings, consisting of a length field followed by that many bytes comprising the label.

Basic Data Formats

Let's step through a simplified Nail grammar for DNS packets, shown in Figure 1. For this grammar, Nail produces the type declarations shown in Figure 2 and the parser and generator functions shown in Figure 3. Nail grammars are reusable between applications, and we will use this grammar to implement both a DNS server and a client, which previously would have had two separate handwritten parsers, leading to bugs such as the Android Master Key.

A Nail grammar file consists of rule definitions—for example, lines 1–20 of Figure 1 assign a name (`dnspacket`) to a grammar production (lines 2–20). If you are not familiar with other parsers, you can imagine rules as C type declarations on steroids (although our syntax is inspired by Go).

```
struct dnspacket {
  uint16_t id;
  uint8_t qr;
  /* ... */
  struct {
    struct question *elem;
    size_t count;
  } questions;
};
```

Figure 2: Portions of the C data structures defined by Nail for the DNS grammar shown in Figure 1

```
struct dnspacket *parse_dnspacket(NailArena *arena,
 const uint8_t *data,
 size_t size);
int gen_dnspacket(NailArena *tmp_arena,
   NailStream *out,
   struct dnspacket *val);
```

Figure 3: The API functions generated by Nail for parsing inputs and generating outputs for the DNS grammar shown in Figure 1

Just as C supports various constructs to build up types, such as structures and unions from pointers and elemental types, Nail supports various *combinators* to represent features of a file or protocol. We will present the features we used in implementing DNS. A more complete reference can be found in [4], with a detailed rationale in [1].

Integers and Constraints. Because Nail is designed to cope with binary formats, it handles not only common integer types (e.g., `uint16`) but bit fields of any length, such as `uint1`. These integers are exposed to the programmer as an appropriately sized machine integer (e.g., `uint8_t`). Nail also supports constraints on integer values, limiting the values to either a range (line 23, `|1..16`), which can optionally be half open or a fixed set (line 24, `|[1,255]`). Both types of constraint can be combined, e.g., `| [1..16,255]`. Constant values are also supported—e.g., line 10: `uint3=0` represents three reserved bits that must be 0. Because constant values carry no information, they are not represented in the data type.

Structures. The body of the `dnspacket` rule is a structure, which contains any number of fields enclosed between curly braces. Each field in the structure is parsed in sequence and represented as a structure to the programmer. Contrary to other programming languages, Nail does not have a special keyword for structs. We also reverse the usual structure-field syntax: `id uint1` is a field called `id` with type `uint1`. Often, Nail grammars have structures with just one non-constant field—for example, when parsing a fixed header. Nail supports this with an alternative form of structures, using angle brackets, that contains one unnamed, non-constant field, which is represented directly in the datatype, without introducing another layer of indirection, as shown on line 40.

Arrays. Nail supports various forms of arrays. Line 40 shows how to parse a domain in a DNS packet with `many`, which keeps repeating the `label` rule until it fails. In the next section, we will explain how to handle count fields, and our full paper describes how to handle various array representations (such as delimiters or non-empty arrays).

Redundant Data

Data formats often contain values that are determined by other values or the layout of information, such as checksums, duplicated information, or offset and length fields. Exposing such values risks inconsistencies that could trick the program into unsafe behavior. Therefore, we represent such values using *dependent fields* and handle them transparently during parsing and generation without exposing them to the application. Dependent fields are handled like other fields when parsing input but are only stored temporarily instead of in the data type. Their value can be referenced by other parsers until it goes out of scope. When generating output, Nail inserts the correct value.

In DNS packets, the packet header contains count fields (`qc, ac, ns, and ar`), which contain the number of questions and answers that follow the header and which we represent by dependent fields (lines 12–15). Dependent fields are defined within a structure like normal fields, but their name starts with an @ symbol. A dependent field is in scope and can be referred to by the definition of all subsequent fields in the same structure. Dependent fields can be passed to rule invocations as parameters.

Nail allows handling count fields with `n_of`, which parses an exact number of repetitions of a rule. Lines 16–19 in Figure 1 show how to use `n_of` to parse the question and answer records in a DNS packet. Other dependencies, such as offset fields or checksums, are not handled directly by combinators but through transformations, as we describe next.

Input Streams and Transformations

So far, we have described a parser that consumes input a byte at a time from beginning to end. However, real-world formats often require nonlinear parsing. Offset fields require a parser to move to a different position in the input, possibly backwards. Size fields require the parser to stop processing before the end of input has been reached. Other cases, such as compressed data and checksums, require more complicated processing on parts of the input before it can be handled.

For a parser to be useful, it needs to support all these ways of structuring a format. This is why data description languages like PADS [2] contain not just a kitchen sink, but a kitchen store

Nail Grammar	External Format	Internal Data Type in C
`uint4`	4-bit unsigned integer	`uint8_t`
`int32 | [1,5..255,512]`	Signed 32-bit integer x ∈{1,5..255,512}	`int32_t`
`uint8 = 0`	8-bit constant with value 0	`/* empty */`
`optional int8 | 16..`	8-bit integer ≥ 16 or nothing	`int8_t *`

`many int8 | ![0]`	A NULL-terminated string	`struct {`
		`  size_t N_count;`
		`  int_t *elem;`
		`};`

`{`
`  hours uint8`
`  minutes uint8`
`}`
	Structure with two fields	`struct {`
		`  uint8_t hours;`
		`  uint8_t minutes;`
		`};`

`<int8='"'; p; int8='"'>`	A value described by parser *p*, in quotes	The data type of *p*

`choose {`
`  A = uint8 | 1..8`
`  B = uint16 | 256..`
`}`
	Either an 8-bit integer between 1 and 8, or a 16-bit integer larger than 256	`struct {`
		`  enum {A, B} N_type;`
		`  union {`
		`    uint8_t a;`
		`    uint16_t b;`
		`  };`
		`};`

`@valuelen uint16`
`value n_of @valuelen uint8`
	A 16-bit length field, followed by that many bytes	`struct {`
		`  size_t N_count;`
		`  uint8_t *elem;`
		`};`

`$data transform`
`  deflate($current @method)`
	Applies programmer-specified function to create new stream	`/* empty */`

`apply $stream p`	Apply parser *p* to stream $stream	The data type of *p*
`foo = p`	Define rule `foo` as parser *p*	`typedef /* type of p */ foo;`
`* p`	Apply parser *p*	Pointer to the data type of *p*

Figure 4: Syntax of Nail parser declarations and the formats and data types they describe

full of features, and a language that can handle all possible formats will be a general purpose programming language. Instead, we keep Nail itself small and introduce an interface that allows complicated format structures to be handled by plugin *transformations* in a general purpose language. Of course, we ship Nail with a handy library of common transformations to handle common format features, such as offsets, sizes, and checksums.

These *transformations* consume and produce streams—sequences of bytes—which can be further passed to other transformations and eventually parsed by a Nail rule. Transformations can also access values in dependent fields. Streams can be subsets of other streams: for example, the substream starting at an offset given in a dependent field to handle pointer fields, or computed at runtime, such as by decompressing another stream with zlib.

Transformations are two arbitrary functions called during parsing and output generation. The parsing function consumes any number of streams and dependent field values, and produces any number of temporary streams. This function may reposition and read from the input streams and read the values of dependent fields, but not change their contents and values. The generating function has to be an inverse of the parsing function, consuming streams and producing dependent field values and other streams.

As a concrete example, we will show a grammar for ProtoZIP, a very simple archive format inspired by ZIP in Figure 5. ProtoZIP consists of a variable-length end-of-file directory, which is a magic number followed by an array of filenames and pointers to compressed files. A grammar for the real ZIP format, which has more layers of indirection, is presented in the full paper.

In Figure 5, the grammar first calls the zipdir transform on line 2, which finds the magic number and splits the file into two streams, one containing the compressed files, the other the directory. Streams are referred to with `$identifiers`, similar to dependent fields. A C prototype of the `zipdir` transform is shown in Figure 6.

When parsing input, this will call `zipdir_parse`, which takes `$current`—an implicit identifier always referring to the stream currently being handled—and returns `$files` and `$header`. When generating output, this will call `zipdir_generate`, which appends `$files` and `$header` to `$current`.

Line 3 of Figure 5 then applies the `dir` rule to the `$header` stream, passing it the `$files` stream. Within `dir`, `$current` is now `$header` and input is parsed from and output generated to that stream. The `dir` rule in turn describes the structure of the directory—a magic number and a count field, followed by that many file descriptors. Each file descriptor is then parsed with two transformations: the standard-library slice, which describes an offset and a size within another stream, and the custom `zlib`, which compresses a stream using zlib. Finally, we apply a trivial grammar (line 14) to the contents.

In a more complicated example, such as an Office document, we could now specify grammars for each entry within an archive.

Transformations need to be carefully written, because they can violate Nail's safety properties and introduce bugs. However, as we will show below (see Applications), Nail transformations are much shorter than handwritten parsers, and many formats can be represented with just the transformations in Nail's standard library. For example, our Zip transformations are 78 lines of code, compared to 1600 lines of code for a handwritten parser. Additionally, Nail provides convenient and safe interfaces for allocating memory and accessing streams that address the most common occurrences of buffer overflow vulnerabilities.

```
1  protozip = {
2    $files, $header transform zipdir($current)
3    contents apply $header dir($files)
4  }
5  dir ($files) = {
6    uint32 =  0x00034b50
7    @count uint32
8    files n_of @count {
9      @off uint32
10     @size uint32
11     filename many (uint8 | ![0])
12     $compr transform slice_u32($files @off @size)
13     $decomp transform zlib($compr)
14     contents apply $decomp (many uint8)
15   }
16 }
```

Figure 5: Nail grammar for ZIP files. Various fields have been cut for brevity.

```
int zip_end_of_directory_parse(
  NailArena *tmp, NailStream *out_files,
  NailStream *out_dir, NailStream *in_current);
int zip_end_of_directory_generate(
  NailArena *tmp, NailStream *in_files,
  NailStream *in_dir, NailStream *out_current);
```

Figure 6: Signatures of stream transform functions for handling the end-to-beginning structure of ProtoZIP files

Using Nail

Real-World Formats

We used Nail to implement grammars for seven protocols with a range of challenging features. Figure 7 summarizes our results. Despite the challenging aspects of these protocols, Nail is able to capture them by relying on its novel features: dependent fields, streams, and transforms. In contrast, state-of-the-art parser generators would be unable to fully handle five out of the seven data formats.

DNS. Previously, we used a grammar for DNS packets shown in Figure 1 to show how to write Nail grammars. This example grammar corresponds almost directly to the diagrams in RFC 1035, which defines DNS. Nail's dependent fields handle DNS's count fields, and transformations represent label compression. At best, both of these features are awkward to handle with existing tools.

ZIP. An especially tricky data format is the ZIP compressed archive format, as specified by PKWARE. At the end of each ZIP file is an *end-of-directory header*. This header contains a variable-length comment, so it has to be located by scanning backwards from the end of the file until a magic number and a valid length field are found. Many ZIP implementations disagree

Protocol	LoC	Challenging Features
DNS packets	48+64	Label compression, count fields
ZIP archives	92+78	Checksums, offsets, variable length trailer, compression
Ethernet	16+0	—
ARP	10+0	—
IP	25+0	Total length field, options
UDP	7+0	Checksum, length field
ICMP	5+0	Checksum

Figure 7: Protocols, sizes of their Nail grammars, and challenging aspects of the protocol that cannot be expressed in existing grammar languages. A + symbol counts lines of Nail grammar code (before the +) and lines of C code for protocol-specific transforms (after the +).

on the exact semantics of this, such as when the comment contains the magic number [6]. This header contains the offset and the size of the *ZIP directory*, which is an array of *directory entry headers*, one for every file in the archive. Each entry stores file metadata in addition to the offset of a *local file header*. The local file header duplicates most information from the directory entry header and is followed immediately by the compressed archive entry. Duplicating information made sense when ZIP files were stored on floppy disks with slow seek times and high fault rates, but nowadays it leads to parsers being confused, such as in the recent Android Master Key bug.

Nail captures these redundancies with dependent fields, eliminating the ambiguities. It also decompresses archive contents transparently with transformations, which allows parsing the contents of an archive file—allowing formats based on ZIP, such as Microsoft Office documents, to be handled with one grammar.

Applications

We implemented two applications—a DNS server and an unzip program—based on the above grammars, and will compare the effort involved and the resulting security to similar applications with handwritten parsers and with other parser generators. We will use lines of code as a proxy for programmer effort. To evaluate security, we will argue how our design avoids classes of vulnerabilities and fuzz-test one of our applications.

DNS. Our DNS server parses a zone file, listens to incoming DNS requests, parses them, and generates appropriate responses. The DNS server is implemented in 183 lines of C, together with 48 lines of Nail grammar and 64 lines of C code implementing stream transforms for DNS label compression. In comparison, Hammer [5] ships with a toy DNS server that responds to any valid DNS query with a CNAME record to the domain “spargelze.it”. Their server consists of 683 lines of C, mostly custom validators, semantic actions, and data structure

Application	LoC w/ Nail	LoC w/o Nail
DNS server	295	683 (Hammer parser)
unzip	220	1,600 (Info-Zip)

Figure 8: Comparison of code size for two applications written in Nail, and a comparable existing implementation without Nail

definitions, with 52 lines of code defining the grammar with Hammer’s combinators. Their DNS server does not implement label compression, zone files, etc.

To evaluate whether Nail-based parsers are compatible with good performance, we compare the performance of our DNS server to that of ISC BIND 9 release 9.9.5, a mature and widely used DNS server. We simulate a load resembling that of an authoritative name server, generating a random zone file and a random sequence of queries, with 10% non-existent domains. We repeated this sequence of queries for one minute against both DNS servers. We found that our DNS server is approximately three times faster than BIND. Although BIND is a more sophisticated DNS server and implements many features that are not present in our Nail-based DNS server and that allow it to be used in more complicated configurations, we believe our results demonstrate that Nail’s parsers are not a barrier to achieving good performance.

ZIP. We implemented a ZIP file extractor in 50 lines of C code, together with 92 lines of Nail grammar and 78 lines of C code implementing two stream transforms (one for the DEFLATE compression algorithm with the help of the zlib library, and one for finding the end-of-directory header). The unzip utility contains a file `extract.c`, which parses ZIP metadata and calls various decompression routines in other files. This file measures over 1,600 lines of C, which suggests that Nail is highly effective at reducing manual input parsing code, even for the complex ZIP file format.

In our full paper [1], we present a study of 15 ZIP parsing bugs. Eleven of these vulnerabilities involved memory corruption during input handling, which Nail’s generated code is immune to by design. We also fuzz-tested our DNS server. More interestingly, Nail also protects against parsing inconsistency vulnerabilities like the four others we studied. Nail grammars explicitly encode duplicated information such as the redundant length fields in ZIP that caused a vulnerability in the Python ZIP library. The other three vulnerabilities exist because multiple implementations of the same protocol disagree on some inputs. Handwritten protocol parsers are not very reusable, as they build application-specific data structures and are tightly coupled to the rest of the code. Nail grammars, however, can be reused between applications, avoiding protocol misunderstandings.

Conclusion

We presented the design and implementation of *Nail,* a tool for parsing and generating complex data formats based on a precise grammar. This helps programmers avoid memory corruption and inconsistency vulnerabilities while reducing effort in parsing and generating real-world protocols and file formats. Nail captures complex data formats by introducing *dependent fields, streams,* and *transforms.* Using these techniques, Nail is able to support DNS packet and ZIP file formats, and enables applications to handle these data formats in many fewer lines of code.

Nail and all of the applications and grammars developed in this paper are released as open-source software, available at https://github.com/jbangert/nail. A more detailed discussion of our design and our results is available in [1].

Acknowledgments

We thank M. Frans Kaashoek, the OSDI reviewers, and K. Park 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] J. Bangert and N. Zeldovich, "Nail: A Practical Tool for Parsing and Generating Data Formats," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pp. 615–628, Broomfield, CO, Oct. 2014, USENIX Association: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/bangert.

[2] K. Fisher and R. Gruber, "PADS: A Domain-Specific Language for Processing Ad Hoc Data," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 295–304, Chicago, IL, June 2005.

[3] G. Hotz, evasi0n 7 writeup, 2013: http://geohot.com/e7writeup.html.

[4] D. Kaminsky, M.L. Patterson, and L. 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*, pp. 289–303, Jan. 2010.

[5] M. Patterson and D. Hirsch, Hammer parser generator, March 2014: https://github.com/UpstandingHackers/hammer.

[6] J. Wolf, "Stupid ZIP file tricks!" in BerlinSides 0x7DD, 2013.