Nail: A practical tool for parsing and generating data formats

Paper #140

ABSTRACT

Nail is a tool that greatly reduces the programmer effort for safely parsing and generating data formats defined by a grammar. Nail introduces several key ideas to achieve its goal. First, Nail uses the protocol grammar to define not just the data format, but also the internal object model of the data. Second, Nail eliminates the notion of semantic actions, used by existing parser generators, which reduces the expressive power but allows Nail to both *parse* data formats and *generate* them from the internal object model, by maintaining a *semantic bijection* between the data format and the object model. Third, Nail introduces *dependent fields* and *stream transforms* to capture protocol features such as size and offset fields, checksums, and compressed data, which are impractical to express in existing protocol languages. Using Nail, we implement an authoritative DNS server in C in under 300 lines of code and grammar, and an `unzip` program in C in 220 lines of code and grammar, demonstrating that Nail makes it easy to parse complex real-world data formats. Performance experiments show that a Nail-based DNS server can outperform the widely used BIND DNS server, demonstrating that systems built with Nail can achieve good performance.

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 image decompression library has had 24 remotely exploitable vulnerabilities from 2007 to 2013 [10], 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 [9].

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, and is still error-prone, for four reasons:

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. Not only does the programmer allocate memory and manipulate pointers, operations that are error-prone, but 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 focus on just parsing an input, rather than producing also an output, thus requiring the programmer to manually construct outputs, which is work-intensive and more code that could contain errors. Some parser generators, such as Boost.Spirit [12], 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, many data formats contain redundancies, such as repeating information in multiple structures. Applications usually do not explicitly check for consistency, and if different applications use different instances of the same value, an attacker can craft input that causes inconsistencies [13]. Furthermore, security vulnerabilities can occur when an application assumes two repetitions of the same data to be consistent, such as allocating a buffer based on the value of one size field and copying into that buffer based on the value of another [1].

Finally, real-world data formats, for example PNG or PDF, are hard to represent with existing parser generators. Those parsers cannot directly deal with length or checksum fields directly, so the programmer has to either write potentially unsafe code to deal with such features, or

build contrived grammar constructs, such as introducing one grammar rule for each possible value of a length field. Offset fields, which specify the position at which some data structure is located, usually require the programmer to manipulate a parser's internal state to re-position its input. More complicated transformations, such as handling compressed data, cannot be represented at all.

This paper presents the design and implementation of Nail, a parser generator that greatly reduces the programmer effort required to use grammars. Nail addresses the above four challenges with several key ideas, as follows.

First, Nail grammars define both a format's external representation and an internal object model. This removes the semantic actions and type declarations that programmers have to write with existing parser generators. While this somewhat reduces the flexibility of the internal model, it forces the programmer to clearly separate syntactic validation and semantic processing.

Second, this well-defined internal representation allows Nail to maintain a *semantic bijection* between data formats and their internal object model. As a result, this enables Nail to not just parse input but also generate output from the internal representation, without requiring the programmer to write additional code.

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 in this category. Transformations allow the programmer to escape the generated code to modify the raw data and interact with dependent fields in a controlled manner.

To evaluate whether Nail's design is effective at handling real-world data formats, we implemented a prototype of Nail for C. Using our prototype, we implemented grammars for parts of an IP network stack, for DNS packets, and for ZIP files, each in about a hundred lines of grammar. On top of these grammars, we were able to build a DNS server in under 200 lines of C code, and an `unzip` utility in about 50 lines of C code, with performance comparable to or exceeding existing implementations. This suggests both that Nail is effective at handling complex real-world data formats, and that Nail makes it easy for application developers to parse and generate external data representations. Performance results show that the Nail-based DNS server outperforms the widely used BIND DNS server, demonstrating that Nail-based parsers and generators can achieve good performance.

The rest of this paper is organized as follows. §2 puts Nail in the context of related work. §3 motivates the need for Nail by examining past data format vulnerabilities. §4 describes Nail's design. §5 discusses our implementation of Nail. §6 provides evaluation results, and §7 concludes.

2 Related work

Parsers. Generating parsers and generators from an executable specification is the core concept of interface generators, such as CORBA [30], XDR [34], and Protocol Buffers [36]. However, interface generators do not allow the programmer to specify the byte-level data format; instead, they define their own data encoding that is specific to a particular interface generator. For instance, XDR-based protocols are incompatible with Protocol Buffers. Moreover, this means that interface generators cannot be used to interact with existing protocols that were not defined using that interface generator in the first place. As a result, interface generators cannot be used to parse or generate widely used formats such as DNS or ZIP, which is a goal for Nail.

Closely related work has been done at Bell Labs with the PacketTypes system [27]. 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 length fields and constraints. PacketTypes also cannot handle complicated encodings such as compressed data, which are supported by Nail's stream transforms.

Parser generators for binary protocols were first introduced by the Hammer [32] parser. While previous parser generators could also be used to write grammars for binary protocols,[1] 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 handled transparently.

Hammer implements grammars as language-integrated parser combinators, an approach popularized by Parsec for Haskell [23]. The parser combinator style (to our knowledge, first described by Burge [5]) is a natural way of concisely expressing top-down grammars [11] by composing them from one or multiple sub-parsers.[2] Hammer then constructs a tree of function pointers which can be invoked to parse a given input into an AST.

Nail improves upon Hammer in three ways. First, Nail generates output in addition to parsing input. Second, Nail does not require the programmer to write potentially insecure semantic actions. Last, Nail's structural dependencies and stream transforms allow it to work with protocols that Hammer cannot handle, such as protocols

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

[2] For more background on the history of expressing grammars, see Bryan Ford's masters thesis [14], which also describes the default parsing algorithm used by Hammer.

with offset fields, length fields, checksums, or compressed data, although Hammer has special facilities for arrays immediately preceded by their length.

Parsifal [24] is a parser framework that also supports generating output for OCaml. Parsifal structures grammars as an OCaml type that holds an internal model and functions for parsing input and output. However, Parsifal can produce parsers and generators only for simple, fixed-size structures. The programmer can then use these when implementing parsers and generators for more complicated formats, manually handling offsets, checksums, and the like, risking bugs. Nail handles more complicated constructs without the programmer manually writing code to support them.

We presented an earlier design of Nail at a workshop [3]. At that stage, Nail had only limited support for dependent fields, and did not support stream transforms at all, which are crucial for supporting real-world formats like DNS and ZIP. The workshop paper also did not provide a detailed design discussion or evaluation.

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 specified with a formal grammar in an executable form. Unfortunately, parser frameworks are seldom used to recognize machine-created input; as we demonstrate in §6, state-of-the-art parser generators are not suitable for parsing or generating many real-world data formats.

A notable exception is the Mongrel web server [33] which uses a grammar for HTTP written in the Ragel regular expression language [35]. Mongrel was re-written from scratch multiple times to achieve better scalability and design, yet the grammar was reused across all iterations [31]. We hope that Nail's ideas make it possible to handle a wider range of protocols using parser generators, and to build more applications on top of grammar-based parsers.

3 MOTIVATION

To motivate the need for Nail, this section presents a case study of vulnerabilities due to ad-hoc input parsing and output generation. Broadly speaking, parsing vulnerabilities can lead to two kinds of problems—memory corruption and logic errors—and as we show, both are prevalent in software and lead to significant security problems.

Widely exploited parsing errors. Three recent high profile security vulnerabilities are due to logic errors in input processing. In all cases, when the vulnerabilities were fixed, a similar flaw was exposed immediately afterwards, showing the need for a different approach to input handling that eliminates those vulnerabilities by design.

The Evasi0n jailbreak for iOS 6 [13] relies on the XNU kernel and user-mode code-signing verifier interpreting executable metadata differently, so the code signature sees different bytes at a virtual address than kernel maps into the process. The next version of iOS added an explicit check for this particular metadata inconsistency. However, because parsing and processing of the input data is still mixed, the jailbreakers could set a flag that re-introduced the ambiguity after the check, but before signatures are verified [18], which allowed iOS 7 to be jailbroken.

Similarly, vulnerabilities in X.509 parsers for SSL certificates allowed attackers to get certificates for domains they do not control. First, Moxie Marlinspike discovered that the X.509 parsers in popular browsers handle NUL-bytes in certificates incorrectly [25]. After this vulnerability was fixed, Dan Kaminsky discovered [22] that other structures, such as length fields and duplicated data, were also handled incorrectly.

Similarly, the infamous Android master key bug [16] completely bypassed Android security by exploiting parser inconsistencies between the ZIP handler that checks signatures for privileged applications and the ZIP implementation that ultimately extracts those files. Thus, privileged application bundles could be modified to include malicious code without breaking their signatures. Google quickly fixed this particular parser inconsistency, but another vulnerability, based on a different inconsistency between the parsers, was quickly disclosed [17].

Case study: ZIP file handling. To understand the impact of parsing mistakes in real-world software, we conducted a systematic study of vulnerabilities related to ZIP file parsing. The ZIP format has been associated with many vulnerabilities, and the PROTOS Genome project [7] found numerous security vulnerabilities in most implementations of ZIP and other archive formats that are directly related to input handling. We extend this study by looking at the CVE database.

We found 83 vulnerabilities in the CVE database [8] that mention the search string "ZIP." Just 16 of these vulnerabilities were related to processing ZIP archives; the rest were unrelated to ZIP archives or involved applications insecurely using the contents of untrusted ZIP files. Figure 1 summarizes the 16 ZIP-related vulnerabilities.

These input processing vulnerabilities fall into two broad classes. The first class, which occurred 11 times[3], 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

[3]We classified the following vulnerabilities as memory corruption attacks based on their description: 2013-5660, - 0742, -0138, 2012-4987, -1163,-1162, 2011-2265, 2010-4535, -1657, -1336, -1218

Classification	Example CVE	Example description	Count
Memory corruption	2013-5660	Buffer overflow	11
Parser ambiguities	2013-1462	Multiple virus scanners interpret ZIP files incorrectly.	4
Semantic misunderstanding	2014-2319	Weak cryptography used even if user selects AES	1
Total vulnerabilities related to .zip processing			**16**

Figure 1: Classification of known vulnerabilities in the CVE database between 2010 and 2014 related to the search term "ZIP" and involving the ZIP file format.

a wide range of techniques, for example static analysis, dynamic instrumentation, and address space layout randomization, that make it more difficult for an adversary to exploit these bugs. Nail helps developers using lower-level languages to avoid these bugs in the first place.

The second class, which occurred four times in our study, is logic errors, where application code misinterprets input data. Safe languages and exploit mitigation technologies do not help against such vulnerabilities. This can lead to serious security consequences when two systems disagree on the meaning of a network packet or a signed message, as shown by the vulnerabilities we described before. CVE-2013-0211 shows that logic errors can be the underlying cause of memory corruption, when one part of a parser interprets a size field as a signed integer and another interprets it as an unsigned integer. CVE-2013-7338 is a logic error that allows an attacker to craft ZIP files that are incorrectly extracted or result in application hangs with applications using a Python ZIP library, because this library does not check that two fields that contain the size of a file contain the same value. The Android ZIP file signature verification bug that we described earlier was also among these 4 vulnerabilities.

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.

4 DESIGN

Nail's goals are to reduce programmer effort required to safely interact with data formats and prevent vulnerabilities like those described in §3. In particular, this means:

- Using the grammar to define both the *external format* and the *internal representation*, allowing the same grammar to be re-used in multiple programs, avoiding vulnerabilities like the Android Master Key.
- Both parsing inputs into internal representations, as well as generating outputs from internal representations, without requiring the programmer to write any semantic actions. This prevents vulnerabilities such as the XNU bug, where format recognition and semantics are mixed and interact in unexpected ways.
- Eliminating redundancy in internal representations, such as storing both an explicit length field and an implicit length of a container data structure, to provide programmers an unambiguous view of the data, to avoid bugs such as in the Python ZIP library.
- Allow programmers to define grammars for complex real-world data formats through well-defined extensibility mechanisms.

4.1 Overview

Internal model. Nail grammars describe both the *external format* and an *internal representation* of a protocol, as opposed to existing grammar languages, which describe only the former and leave it up to the programmer to construct an internal representation, typically in the form of an abstract syntax tree. Nail produces the following from a single, descriptive grammar:

- *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.

Semantic bijection. In order to generate output, Nail maintains a *semantic bijection* between the external format and the internal model. That is, when Nail parses an input into an internal representation, and then generates output from that representation, the two byte streams (input and output) will have the same meaning (i.e., be interpreted equivalently by Nail). However, the byte streams might not be identical. A bijection in the traditional sense often does not make sense for data formats. Consider a grammar for a text language that tolerates white space, or a binary protocol that tolerates arbitrarily long padding. 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*. Similarly, Nail does not preserve the layout of objects referred to by their offsets. If the grammar consists of a simple protocol without offset fields, constants, and the like, there is a conventional bijection between internal models and valid parser inputs.

Hide redundant information. Nail's internal model is designed to hide unneeded and redundant information from the application. Nail introduces *dependent fields*, which contain data that can be computed during generation and need to be kept as additional state during parsing. Dependent fields are, for example, used to represent lengths, offsets, and checksums.

Parser extensions. Real-world protocols contain complicated ways of encoding data. Fully representing these in an intentionally limited model such as our parser language is impractical. Therefore, Nail introduces *transformations*, which allow arbitrary code by the programmer to interact with the parser and generator. Nail parsers and generators interact with data through an abstract stream, which allows reading and writing of bits as re-positioning. Transformations allow the programmer to write functions in a general purpose language that consume streams and define new temporary streams, while also reading or writing the values of dependent fields.

Initial versions of Nail's design included a special combinator for handling offset fields, which consumed a dependent field and applied a parser at the offset specified therein. However, it proved impossible to foresee all the ways in which a protocol could encode an offset; for example, some protocols such as PDF and ZIP locate structures by scanning for a magic number starting at the end of the file or a fixed offset. In nested grammars, offsets are also not necessarily computed from the beginning of a file or packet. Nail's transformations allow the programmer to write arbitrary functions that can handle such structures and streams, which are a generic abstraction for input and output data that allow the decoded data to be integrated with the rest of the generated Nail parser.

4.2 Basics

A Nail parser defines both the structure of some external format and a data type to represent that format. Parsers are constructed by combinators over simpler parsers, an approach popularized by the Parsec framework [23]. We provide the most common combinators familiar from other parser combinator libraries, such as Parsec and Hammer [32] and extend them so they also describe a data type.

We present both a systematic overview of Nail's syntax with short examples in Figure 2, and explain our design in more detail below, using a grammar for the well-known DNS protocol as a running example (shown in Figure 3).

Rules. A Nail grammar consists of rules that assign a parser to a name. Rules are written as assignments, such as `ints = /*parser definition*/`, which defines a rule called `ints`. As we will describe later in §4.3 and §4.4, rules can optionally consume parameters. Rules can be invoked in a Nail grammar anywhere a parser can appear.

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

Figure 3: Nail grammar for DNS packets, used by our prototype DNS server.

Rule invocations act as though the body of the rule had been substituted in the code. If parameters appear, they are passed by reference.

Integers and constraints. Nail's fundamental parsers represent signed or unsigned integers with arbitrary lengths up to 64 bits. Note that is possible to define parsers for sub-byte lengths, for example, the flag bits in the DNS message header, in lines 4 through 8.

The grammar can also constrain the values of an integer. Nail expresses constraints as a set of permissible values or value ranges. Extending the Nail language and implementation to support richer constraints languages would be relatively trivial, however we have found that the current syntax covers permissible values within existing protocols

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 $x \in \{1, 5..255, 512\}$	`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 (§4.4)	`/* empty */`
`apply $stream p`	Apply parser *p* to stream `$stream` (§4.4)	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 2: Syntax of Nail parser declarations and the formats and data types they describe.

correctly and concisely.

Repetition. The *many* combinator takes a parser and applies it repeatedly until it fails, returning an array of the inner parser's results. In line 39 of the DNS grammar, a sequence of labels is parsed by parsing as many labels as possible, that is, until an invalid length field is encountered. 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. This is useful for parsing an array of items delimited by a separator.

Structures. Nail provides a structure combinator with semantic labels instead of the sequence combinator that other parser combinator libraries use to capture structures in data formats. The structure combinator consists of a sequence of fields, typically consisting of a label and a parser that describes the contents of that field, surrounded by curly braces. Other field types will be described below. The syntax of the structure combinator is inspired by the Go language [19], with field names preceding their definition.

Constants. In some cases, not all bytes in a structure actually contain information, such as magic numbers or reserved fields. Those fields can be represented in Nail grammars by constant fields in structures. Constant fields do not correspond to a field in the internal model, but they are validated during parsing and generated during output. Constants can either have integer values, such as in line 9 of the DNS grammar, or string values for text-based protocols, e.g. `many uint8 = "Foo"`.

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. Nail therefore allows repeated constants, such as `many (uint8=' ')`, which parses any number of space characters, or `|| uint8 = 0x90 || uint16 = 0x1F0F`, which parses two of the many representations for x86 NOP instructions, which are used as padding between basic blocks in an executable.

As discussed above, choosing to use these combinators on constant parsers weakens the bijection between the format and the data type, as there are multiple byte-strings that correspond to the same internal representation and the generator chooses one of these.

Wrap combinator. When implementing real protocols with Nail, we often found structures that consist of many constants 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 external format is defined as though the wrap combinator were a structure, but the data model does not introduce a structure with just one element, making the application-visible representation (and thus application code) more concise. Line 39 of the DNS grammar uses the wrap combinator to hide the terminating NUL-byte of a sequence of labels.

Choices. If multiple structures can appear at a given position in a format, the programmer lists the options along with a label for each in the `choose` combinator. During parsing, Nail remembers the current input position and attempts each option in the order they appear in the grammar. If an option fails, the parser backtracks to the initial position. If no options succeed, the entire combinator fails. In the data model, choices are represented as tagged unions. The programmer has to be careful when options overlap, because if the programmer meant to generate output for a choice, but the external representation is also valid for an earlier, higher-priority option, the parser will interpret it as such. However, real data formats normally do not have this overlap and we did not encounter it in the grammars we wrote. An example is provided in Figure 4.

Optional. Nail includes an `optional` combinator, which attempts to recognize a value, but succeeds without consuming input when it cannot recognize that value. Syntactically, `optional` is equivalent to a choice between the parser and an empty structure, but in the internal model it is more concisely represented as a reference that is null when the parser fails. For example, the grammar for Ethernet headers uses `optional vlan_header` to parse the VLAN header that only appears in Ethernet packets transmitted to a non-default VLAN.

```
expr = choose {
  PAREN = <uint8='('; *expr; uint8=')'>
  PRODUCT = sepBy1 uint8='*' expr
  SUM = sepBy1 uint8='+' expr
  INTEGER = many1 uint8 | '0' .. '9'
}
```

Figure 4: Grammar for sums and products of integers.

References. Rules allow for recursive grammars. To support recursive data types, we allow introduce the reference combinator `*` that does not change the syntax of the external format described, but introduces a layer of indirection, such as a reference or pointer, to the model data type. The reference combinator does not need to used when another combinator, such as `optional` or `many`, already introduces indirection in the data type. An example is shown in Figure 4.

4.3 Dependent fields

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. We represent such values using *dependent fields* and handle them transparently during parsing and generation without exposing them to the internal model.

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.

Dependent fields are handled like other fields when parsing input, but their values are not stored in the internal data type. Instead the value can be referenced by subsequent parsers and it discarded when the field goes out of scope. When generating output, Nail visits a dependent field twice. First, while generating the other fields of a structure, the generator reserves space for the dependent field in the output. Once the dependent field goes out of scope, the generator writes the dependent field's value to this space.

Nail provides only one built-in combinator that uses dependent fields, `n_of`, which acts like the `many` combinator, except it represents an exact number, specified in the dependent field, of repetitions, as opposed as many repetitions as possible. For example, DNS labels, which are encoded as a length followed by a value are described in line 38 of the DNS grammar. Other dependencies, such as offset fields or checksums, are not handled directly by combinators, but through transformations, as we describe next.

4.4 Input streams and transformations

Traditional parsers handle input one symbol at a time, from beginning to end. However, real-world formats often require non-linear 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, and perhaps resume executing a parent parser. Other cases, such as compressed data, require more complicated processing on parts of the input before it can be handled.

Nail introduces two concepts to handle these challenges, *streams* and *transformations*. Streams represent a sequence of bytes that contain some external format. The parsers and generators that Nail generates always operate on an implicit stream named `$current` that they process front to back, reading input or appending output. Grammars can use the `apply` combinator to parse or generate external data on a different stream, inserting the result in the data model.

Streams are passed as arguments to a rule or defined within the grammar through *transformations*. The current stream is always passed as an implicit parameter.

Transformations are two arbitrary functions called during parsing and output generation. The parsing function takes any number of stream arguments 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. It takes the same number of temporary streams that the parsing function produces, and writes the same number of streams and dependent field values that the parsing function consumes.

Typically, the top level of most grammars is a rule that takes only a single stream, which may then be broken up by various transformations and passed to sub-rules, which eventually parse various linear fragment streams. Upon parsing, these fragment streams are generated and then combined by the transforms.

To reduce both programmer effort and the risk of unsafe operations, Nails provides implementations of transformations for many common features, such as checksums, size, and offset fields. Furthermore, Nail provides library functions that can be used to safely operate on streams, such as splitting and concatenation. Nail implements streams as iterators, so they can share underlying buffers and can be efficiently duplicated and split.

Transformations need to be carefully written, because they can escape Nail's security and introduce bugs. However, as we will show in §6.2, Nail transformations are much shorter than hand-written parsers and many formats can be represented with just the provided transforms

Transformations can handle a wide variety of patterns in data formats, including the following:

Offsets. A built-in transformation for handling offset fields, which is invoked as follows: `$fragment transform offset_u32($current, @offset)`. This transformation corresponds to two functions for parsing and generation, as shown in Figure 5. It defines a new stream `$fragment` that can be used to parse data at the offset contained in `@offset`, by using `apply $fragment some_parser`.

```
int offset_u32_parse(NailArena *tmp,
  NailStream *out_str, NailStream *in_current,
  const uint32_t *off)
{
  /* out_str = suffix of in_current
               at offset *off */
}

int offset_u32_generate(NailArena *tmp,
  NailStream *in_fragment,
  NailStream *out_current, uint32_t *off)
{
  /* *off = position of out_current */
  /* append in_fragment to out_current */
}
```

Figure 5: Pseudocode for two functions that implement the offset transform.

Sizes. A similar transformation handles size fields. Just like the offset transform, it takes two parameters, a stream and a dependent field, but instead of returning the suffix of the current stream after an offset, it returns a slice of the given size from the current stream starting at its current position. When generating, it appends the fragment stream to the current stream and writes the size of the fragment to the dependent field.

Compressed data. Encoded, compressed, or encrypted data can be handled transparently by writing a custom transformation that transforms a coded stream into one that can be parsed by a Nail grammar and vice versa. This transformation must be carefully written to not have bugs.

Checksums. Checksums can be verified and computed in a transformation that takes a stream and a dependent field. In some cases, a checksum is calculated over a buffer that contains the checksum itself, with the checksum being set to some particular value. Because the functions implementing a transformation are passed a pointer to any dependent fields, the checksum function can set the checksum's initial value before calculating the checksum over the entire buffer, including the checksum.

A real-world example with many different transforms, used to support the ZIP file format, is described in §6.1.

5 IMPLEMENTATION

The current prototype of the Nail parser generator supports the C programming language. The implementation parses Nail grammars with Nail itself, using a 130-line Nail grammar feeding into a 2,000-line C++ program that emits the parser and generator code. Bootstrapping is performed with a subset of the grammar implemented using conventional grammars. And option for C++ STL data models is in 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 recognize the input, and the second to bind this data to the internal model. Currently the parser uses a straightforward top-down algorithm, which can perform poorly on grammars that backtrack heavily. However, preparations have been made to add Packrat parsing [15] that achieve linear time even in the worst case.

Defense-in-Depth. Security exploits often rely on raw inputs being present in memory, for example to include shell-code or crafted stack frames for ROP-attacks in padding fields [4] or the application executing a controlled sequence of heap allocations and de-allocations to place specific data at predictable addresses [21, 26]. Because the rest of the application or even Nail's generated code may contain memory corruption bugs, Nail carefully handles memory allocations as defense-in-depth to make exploiting such vulnerabilities harder.

When parsing input, Nail uses two separate memory arenas. These arenas allocate memory from the system allocator in large, fixed size blocks. Allocations are handled linearly and all data in the arena is zeroed and freed at the same time. Nail uses one arena for data used only during parsing, including dependent fields and temporary streams and is released before the Nail's parser returns. The other arena is used to allocate the internal data type returned and is freed by the application once it is done processing an input.

Furthermore, the internal representation does not include any references to the input stream, which can therefore be zeroed immediately after the parser succeeds, so an attacker has to write an exploit that works without referencing data in the raw input.

6 EVALUATION

In our evaluation of Nail, we answer four questions:

- Can Nail grammars support real-world data formats, and are Nail's techniques critical to handling these formats?
- How much programmer effort is required to build an application that uses Nail for data input and output?
- Does using Nail for handling input and output improve application security?
- Does Nail achieve acceptable performance?

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 6: 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 +).

6.1 Data formats

To answer the first question, we used Nail to implement grammars for seven protocols with a range of challenging features. Figure 6 summarizes these protocols, the lines of code for their Nail grammars, and the challenging features that make the protocols difficult to parse with state-of-the-art parser generators. We find that despite the challenging aspects of these protocols, Nail is able to capture the protocols, 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 5 out of the 7 data formats. In the rest of this subsection, we describe the DNS and Zip grammars in more detail, focusing on how Nail's features enable us to support these formats.

DNS. In Section 4, we introduced Nail's syntax with a grammar for DNS packets, shown in Figure 7. The grammar corresponds almost directly to the diagrams in RFC 1035, which defines DNS [28: §4]. 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.

One challenging aspect of DNS packets lies in the count fields (`qc`, `ac`, `ns`, and `ar`), which represent the number of questions or answers in another part of the packet. Nail's `n_of` combinator handles this situation easily, which would have been difficult to handle for other parsers.

Another challenging aspect of DNS is label compression [28: §4.1.4]. Label compression is used to reduce the

```
int dnscompress_parse(NailArena *tmp,
  NailStream *out_decomp,
  NailStream *in_current);

int dnscompress_generate(NailArena *tmp,
  NailStream *in_decomp,
  NailStream *out_current);
```

Figure 7: Signatures of stream transform functions for handling DNS label compression.

size overhead of including each domain name multiple times in a DNS reply (once in the question section, and at least once in the response section). If a domain name suffix is repeated, instead of repeating that suffix, the DNS packet may write a two-bit marker sequence followed by a 14-bit offset into the packet, indicating the position of where that suffix was previously encoded.

Handling label compression in existing tools, such as Bison or Hammer, would at best be very awkward, because some ad-hoc trick would have to be used to reposition the parser's input stream. Keeping track of the position of all recognized labels would not be enough, as the offset field may refer to any byte within the packet, not just the beginning of labels. For this reason, the DNS server used as the example for Hammer does not support compression.

In contrast, Nail is able to handle label compression, by using a stream transform; the signatures of the two transform functions are shown in Figure 7. When parsing a packet, this transform decompresses the DNS label stream by following the offset pointers. When generating a packet, this transform receives the current suffix as an input, and scans the packet so far for previous occurrences, which implements compression.

ZIP files. An especially tricky data format is the ZIP compressed archive format [20]. ZIP files are normally parsed end-to-beginning. 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 is found. Many ZIP implementations disagree on how to find this header in confusing situations, such as when the comment contains the magic number [38].

This end-of-directory header contains the offset and 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, such as file name, compressed and uncompressed size, and a checksum, in addition to the offset of a *local file header*. The local file header duplicates most information from the directory entry header. The compressed file contents follow the header immediately.

Duplicating information made sense when ZIP files were stored on floppy disks with slow seek times and high fault rates, and memory constraints made it impossible to keep the ZIP directory in memory or the archive was split across multiple disks. However, care must be taken that the metadata is consistent. For example, vulnerabilities could occur if the length in the central directory is used to allocate memory and the length in the local directory is used to extract without checking that they are equal first, as was the case in the Python ZIP library [1]. Figure 8 shows an abbreviated version of our ZIP file grammar. The ZIP grammar is a good example of how transformations capture complicated syntax in a real-world file format; existing parser languages cannot handle a file format of this complexity.

The `zip_file` grammar first splits the entire file stream into two streams based on the `zip_end_of_directory` transform on line 2. `zip_end_of_directory_parse` finds the end-of-directory header as described above, by scanning the file backwards, and splits the file into two streams, one containing the end-of-directory header and one containing the file contents. The `end_of_directory` rule is then applied to the header stream in line 4. All offsets in the ZIP file refer to the beginning of the file, so the stream `$file` which contains the file contents without the header is passed as an argument to all parsers from hereon.

The directory header contains the offset and size of the ZIP directory (lines 9 and 10). The `offset` and `size` transformations extract a stream containing just the directory from the file contents. This stream is then parsed as an array of directory entries in line 17. Each directory entry in turn points to a local file header, which is similarly extracted and parsed with the `file` rule.

The `file` rule starting at line 41, describing a ZIP file entry, takes dependent field parameters containing file metadata information from the directory header. However, this same information is duplicated in the file entry, so the grammar uses the Nail-supplied `u16_depend` transform to check whether the two values are equal. Unlike most other transforms, `u16_depend` does not consume or produce strings; it only checks that two dependent fields are equal when parsing, and assigns the value of the second field to the first when generating. This ensures that the programmer does not have to worry about inconsistencies when handling the internal representation of a ZIP file.

Immediately following the file entry is the compressed data. Because most compression algorithms operate on unbounded streams of data, Nail decompresses data in two steps. First, it isolates the compressed data from the rest of the stream by using the `size` transform, which operates on the current stream, meaning it will consume

```
zip_file = {
  $file, $header transform
    zip_end_of_directory ($current)
  contents apply $header
    end_of_directory ($file)
}
end_of_directory ($file) = {
  // ...
  @directory_size uint32
  @directory_start uint32
  $dirstr1 transform
    offset_u32 ($filestream @directory_start)
  $directory_stream transform
    size_u32 ($dirstr1 @directory_size)
  @comment_length uint16
  comment n_of @comment_length uint8
  files apply $directory_stream n_of
    @total_directory_records dir_entry ($file)
}
dir_entry ($file) = {
  // ...
  @compression_method uint16
  mtime uint16
  mdate uint16
  @crc32 uint32
  @compressed_size uint32
  @uncompressed_size uint32
  @file_name_len uint16
  @extra_len uint16
  @comment_len uint16
  // ...
  @off uint32
  filename n_of @file_name_len uint8
  extra_field n_of @extra_len uint8
  comment n_of @comment_len uint8
  $content transform offset_u32 ($file @off)
  contents apply $content
    file (@crc32, @compression_method,
          @compressed_size, @uncompressed_size)
}
file (@crc32 uint32, @method uint16,
      @compressed_size uint32,
      @uncompressed_size uint32) = {
  uint32 = 0x04034b50
  version uint16
  flags file_flags
  @method_lcl uint16
  // ...
  $compressed transform
    size_u32 ($current @compressed_size)
  $uncompressed transform
    zip_compression ($compressed @method)
  transform crc_32 ($uncompressed @crc32)
  contents apply $uncompressed many uint8
  transform u16_depend (@method_lcl @method)
  // ...
}
```

Figure 8: Nail grammar for ZIP files. Various fields have been cut for brevity.

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

Figure 9: Comparison of code size for three applications written in Nail,and a comparable existing implementation without Nail.

data starting at the current position of the parser in the input. Second, Nail invokes a custom `zip_compression` transform that implements the appropriate compression and decompression functions based on the specified compression method. These functions are otherwise oblivious to the layout or metadata of the file.

6.2 Programmer effort

To evaluate how much programmer effort is required to build an application that uses Nail, we implemented two applications—a DNS server, and an `unzip` program—based on the above grammars, and compared code size with comparable applications that process data manually, using `sloccount` [37]. We also compare the code size of our DNS server to a DNS server written using the Hammer parsing framework, although it does not fully support DNS (e.g., it lacks label compression, among other things). Figure 9 summarizes the results.

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 [32] 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 definitions, with 52 lines of code defining the grammar with Hammer's combinators. Their DNS server does not implement label compression, zone files, etc. From this, we conclude that Nail leads to much more compact code for dealing with DNS packet formats.

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).

Because more recent versions of ZIP have added more features, such as large file support and encryption, the closest existing tool in functionality is the historic version 5.4 of the Info-Zip unzip utility [2] that is shipped with most Linux distributions. The entire unzip distribution is about 46,000 lines of code, which is mostly

optimized implementations of various compression algorithms and other configuration and portability code. However, unzip isolates the equivalent of our Nail tool in the file `extract.c`, which parses the 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.

6.3 Security

We use a twofold approach to evaluate the security of applications implemented with Nail. First, we analyze a list of CVE's related to the ZIP file format and argue how our ZIP tools based on Nail are immune against those vulnerability classes. Second, we present the results of fuzz-testing our DNS server.

ZIP analysis. In §3, we found 15 input handling vulnerabilities related to ZIP files.

11 of these vulnerabilities involved memory corruption during input handling. Because Nail's generated code checks offsets before reading and does not expose any untrusted pointers to the application, it is immune against memory corruption attacks by design.

Nail also protects against parsing ambiguity vulnerabilities like the four others we studied. First, Nail grammars explicitly encode duplicated information such as the redundant length fields in ZIP that caused a vulnerability in the Python ZIP library. Second, the other three vulnerabilities exist because multiple implementations of the same protocol disagree on some inputs. Hand-written 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 re-used between applications, avoiding protocol misunderstandings.

DNS fuzzing. To provide additional assurance that Nail parsers are free of memory corruption attacks, we ran the DNS fuzzer provided with the Metasploit framework [29] on our DNS server, which sent randomly corrupted DNS queries to our server for 4 hours, during which it did not crash or trigger the stack or heap corruption detector.

6.4 Performance

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 [6], a mature and widely used DNS server. We simulate a load resembling that of an authoritative name server. First, we generate domain names consisting of one or two labels randomly selected from an English dictionary, and one label that is one of three popular top-level domains (`com`, `net`, and `org`). Second, we randomly selected 90% of these domains and created a zone file that mapped these domain names to `127.0.0.1`. Finally, we used the

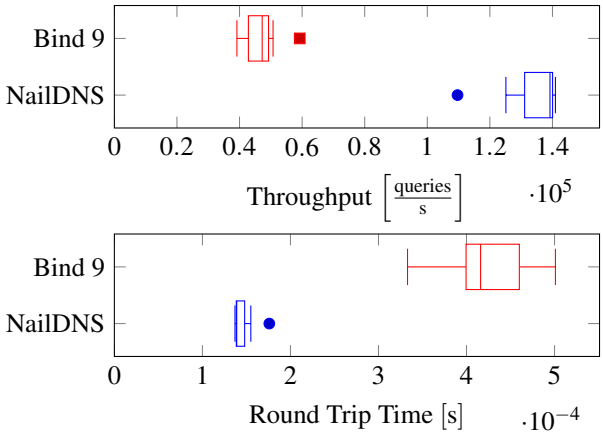

Figure 10: A box plot comparing the performance of the Nail-based DNS server compared to BIND 9.5.5 on 50,000 domains. The boxes show the interquartile range, with the middle showing the median result. The dots show outliers.

`queryperf` tool provided with BIND to query each domain between zero and three times, using a DNS server running on the local machine. We used a single core of an Intel i7-3610QM system with 12GB of RAM. The benchmark tool kept at most 20 queries outstanding at once, and was configured to repeat the same randomized sequence of queries for one minute. We repeated each test seven times with 50,000 domain names, restarting each daemon in between; we also repeated the tests with 1 million domain names, and found similar results. We also performed one initial dry run to warm the file system cache for the zone file.

The results are shown in Figure 10, and demonstrate that our Nail-based DNS server can achieve higher performance and lower latency than BIND. Although BIND is a more sophisticated DNS server, and implements many features that are not present in our Nail-based DNS server, we believe our results demonstrate that Nail's parsers are not a barrier to achieving good performance.

7 Conclusion

This paper presented the design and implementation of *Nail*, a tool for parsing and generating complex data formats based on a precise grammar. Nail helps programmers 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. 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.

REFERENCES

[1] Cve-2013-7338. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-7338.

[2] Infozip. http://www.info-zip.org/.

[3] Anonymous. Omitted for anonymous submission.

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

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

[6] Internet Systems Consortium. Bind 9 dns server, 2005. http://www.isc.org/downloads/bind/.

[7] PROTOS Project Consortium. Protos genome test suite c10-archive. Technical report, University of Oulu, 2007. https://www.ee.oulu.fi/research/ouspg/PROTOS_Test-Suite_c10-archive.

[8] MITRE Corporation. Common vulnerabilities and exposures (CVE). http://http://cve.mitre.org/.

[9] CVE Details. Gnu zlib: List of security vulnerabilities. http://www.cvedetails.com/vulnerability-list/vendor_id-72/product_id-1820/GNU-Zlib.html.

[10] CVE Details. Libpng: Security vulnerabilities. http://www.cvedetails.com/vulnerability-list/vendor_id-7294/Libpng.html.

[11] 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.

[12] 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/.

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

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

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

[16] Jay Freeman. Exploit (& fix) android "master key", 2013. http://http://www.saurik.com/id/17.

[17] Jay Freeman. Yet another Android master key bug, 2013. http://www.saurik.com/id/19.

[18] George Hotz. evasi0n 7 writeup. http://geohot.com/e7writeup.html.

[19] Google Inc. *The Go Programming Language*, May 2014. http://golang.org/doc/.

[20] PKWARE Inc. *.ZIP File Format Specification*, 6.3.3 edition, September 2012. http://www.pkware.com/documents/casestudies/APPNOTE.TXT.

[21] jp. Advanced Doug Lea's malloc Exploits. *Phrack* 61:6. http://phrack.org/issues.html?issue=61&id=6.

[22] 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.

[23] 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.

[24] Olivier Levillain. Parsifal: a pragmatic solution to the binary parsing problem. In *LangSec Workshop at IEEE Security and Privacy*. IEEE, May 2014.

[25] Moxie Marlinspike. More tricks for defeating SSL in practice. *Black Hat USA*, 2009.

[26] MaXX. Vudo malloc Tricks. *Phrack* 57:8. http://phrack.org/issues.html?issue=57&id=8.

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

[28] P. Mockapetris. Domain names – implementation and specification. RFC 1035, Network Working Group, November 1987.

[29] HD Moore et al. The metasploit project, 2009.

[30] Object Management Group, Inc. CORBA FAQ. http://www.omg.org/gettingstarted/corbafaq.htm.

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

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

[33] Zed Shaw. Mongrel HTTP server, 2008. http://www.rubyforge.org/projects/mongrel/.

[34] R. Srinivasan. XDR: External data representation standard. RFC 1832, Network Working Group, August 1995.

[35] 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.

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

[37] David A. Wheeler. Sloccount. http://www.dwheeler.com/sloccount/.

[38] Julia Wolf. Stupid zip file tricks! In *BerlinSides 0x7DD*, 2013.