A Survey of Binary Protocol Parsers

Andrew Fryer[1], Thomas R. Dean[1]

[a]*Department of Electrical and Computer Engineering, Queen's University at Kingston, Kingston, Canada*

Abstract

Effective fuzzing of black-box systems requires knowledge of the structure of the input data that will be sent to the system under test. Decades of research has produced formal language theory and highly optimized parsing algorithms for text input. However, the nature of binary input data presents a different set of problems. Binary files and protocols are often context dependent, and tokenization is ambiguous. We present a comparison of the features and limitations of several popular or otherwise interesting binary parsing frameworks and discuss the implications they may have on designing grammar-based fuzzing systems.

Keywords: Bianry Parsing, Fuzzing
PACS: 0000, 1111
2000 MSC: 0000, 1111

1. Introduction

Many modern services and network APIs are built on top of HTTP(S)[1] using XML[2] or JSON[3] which are all text based protocols. That is, with the exception of binary blob data such as images, most of the information is converted to text before being exchanged between nodes. Several of the core network protocols such as Domain Name System (DNS)[4], and protocols used in industrial control networks, and command and control networks are binary protocols. In this case the data is transmitted directly in binary form over the network.

Network debugging, deep packet inspection for intrusion detection and penetration testing of applications that use binary protocols require parsers to unmarshall the binary messages to an internal form. Some protocols such as Simple Network Management Protocol (SNMP)[5] and Common Object Request Broker Architecture (CORBA)[6] are layered on top of standard encoding protocols such as Abstract Syntax Notation One (ASN.1)[7], Common Data Representation(CDR)[8], or ISO 8211[9]. The standard encoding protocols provide either standard libraries or a specification language (e.g. Interface Description Language (IDL)[10]) for the protocols.

Others such as DNS or Samba use a custom encoding of messages. In many cases, the parsers for these protocols are manually written (e.g. the popular network protocol analyzer Wireshark[11]).

In this paper we survey several existing binary parsing frameworks: ANother Tool for Language Recognition (ANTLR)[12], Kaitai Struct[2], Construct[3], FormatFuzzer [13], Spicy[14], and Structure and Context-Sensitive Language (SCL)[15]. We use the DNS protocol as a basis of comparison and use small samples of the DNS protocol specification to illustrate differences between the parsers.

Section 2 provides some background, including a description of the DNS protocol as well as related research. As it seemed that a comparison structured by parsing framework would switch continuously between topics, properties of parsers are first discussed in Section 4 along with the implications that these have when implementing the DNS protocol. Then, Section 5 provides a comparison of the parsers. Section 6 summarizes the features and limitations of the frameworks. Lastly, Section 7 reflects on this comparison.

2. Background and Related Work

2.1. Standard Encoding Protocols

There are several standard protocols that are used to encode messages.

ASN.1 uses nested tag, length and value fields that allows libraries to read and generate a tree structure from a binary message. The application can then assign meanings to the nodes in the tree. ASN.1 provides several primitive

*This research is result of research funded by Natural Science and Engineering Council of Canada and the Department of National Defence

Email addresses: andrew.fryer@queensu.ca (Andrew Fryer), tom.dean@queensu.ca (Thomas R. Dean)

[1]https://www.antlr.org/
[2]https://kaitai.io/
[3]https://construct.readthedocs.io/en/latest/
[4]https://uds-se.github.io/FormatFuzzer/
[5]https://docs.zeek.org/projects/spicy/en/latest/index.html
[6]https://github.com/thomasrdean/SCLParserGenerator/

Graphical Abstract

A Survey of Binary Protocol Parsers[⋆]

Andrew Fryer, Thomas R. Dean

Highlights

A Survey of Binary Protocol Parsers

Andrew Fryer, Thomas R. Dean

- Research highlight 1
- Research highlight 2

A Survey of Binary Protocol Parsers

Andrew Fryer[1], Thomas R. Dean[1]

[a]*Department of Electrical and Computer Engineering, Queen's University at Kingston, Kingston, Canada*

Abstract

Effective fuzzing of black-box systems requires knowledge of the structure of the input data that will be sent to the system under test. Decades of research has produced formal language theory and highly optimized parsing algorithms for text input. However, the nature of binary input data presents a different set of problems. Binary files and protocols are often context dependent, and tokenization is ambiguous. We present a comparison of the features and limitations of several popular or otherwise interesting binary parsing frameworks and discuss the implications they may have on designing grammar-based fuzzing systems.

Keywords: Bianry Parsing, Fuzzing
PACS: 0000, 1111
2000 MSC: 0000, 1111

1. Introduction

Many modern services and network APIs are built on top of HTTP(S)[1] using XML[2] or JSON[3] which are all text based protocols. That is, with the exception of binary blob data such as images, most of the information is converted to text before being exchanged between nodes. Several of the core network protocols such as Domain Name System (DNS)[4], and protocols used in industrial control networks, and command and control networks are binary protocols. In this case the data is transmitted directly in binary form over the network.

Network debugging, deep packet inspection for intrusion detection and penetration testing of applications that use binary protocols require parsers to unmarshall the binary messages to an internal form. Some protocols such as Simple Network Management Protocol (SNMP)[5] and Common Object Request Broker Architecture (CORBA)[6] are layered on top of standard encoding protocols such as Abstract Syntax Notation One (ASN.1)[7], Common Data Representation(CDR)[8], or ISO 8211[9]. The standard encoding protocols provide either standard libraries or a specification language (e.g. Interface Description Language (IDL)[10]) for the protocols.

Others such as DNS or Samba use a custom encoding of messages. In many cases, the parsers for these protocols are manually written (e.g. the popular network protocol analyzer Wireshark[11]).

In this paper we survey several existing binary parsing frameworks: ANother Tool for Language Recognition (ANTLR)[12][1], Kaitai Struct[2], Construct[3], FormatFuzzer [13][4], Spicy[14][5], and Structure and Context-Sensitive Language (SCL)[15][6]. We use the DNS protocol as a basis of comparison and use small samples of the DNS protocol specification to illustrate differences between the parsers.

Section 2 provides some background, including a description of the DNS protocol as well as related research. As it seemed that a comparison structured by parsing framework would switch continuously between topics, properties of parsers are first discussed in Section 4 along with the implications that these have when implementing the DNS protocol. Then, Section 5 provides a comparison of the parsers. Section 6 summarizes the features and limitations of the frameworks. Lastly, Section 7 reflects on this comparison.

2. Background and Related Work

2.1. Standard Encoding Protocols

There are several standard protocols that are used to encode messages.

ASN.1 uses nested tag, length and value fields that allows libraries to read and generate a tree structure from a binary message. The application can then assign meanings to the nodes in the tree. ASN.1 provides several primitive

*This research is result of research funded by Natural Science and Engineering Council of Canada and the Department of National Defence

Email addresses: andrew.fryer@queensu.ca (Andrew Fryer), tom.dean@queensu.ca (Thomas R. Dean)

[1]https://www.antlr.org/
[2]https://kaitai.io/
[3]https://construct.readthedocs.io/en/latest/
[4]https://uds-se.github.io/FormatFuzzer/
[5]https://docs.zeek.org/projects/spicy/en/latest/index.html
[6]https://github.com/thomasrdean/SCLParserGenerator/

types, including `INTEGER`, `REAL`, `OCTET STRING`, various date formats and others. It also provides predefined aggregate types such as `SEQUENCE` and `SET`. Listing 1 shows a simple record type called `HOUSE` in ASN.1. It consists of an integer `number`, an octet string `street`, and a field, `front`, of another type, `DOOR`. Assuming that the value of `number` is 19, and `street` is 'UNION' and that the type `DOOR` is also a `SEQUENCE` type, it will be encoded as shown in Figure 1.

```
1 HOUSE ::= SEQUENCE {
2     number   INTEGER
3     street   OCTET STRING
4     front    DOOR
5 }
```

Listing 1: Sample ASN.1 Specification

In this example, we are using Basic Encoding Rule (BER). ASN.1 supports several other similar encoding rules such as Distinguished Encoding Rules (DER) and Packet Encoding Rules (PER)[7]. The tag for the integer type is 2, octet string is 4 and sequence is 16. The first byte (16) identifies the message as a sequence that is 32 bytes long in accordance with the second byte. The first element is an integer (2) that is 1 byte long had has the value 19. This is followed by an octet string (4) that is 5 bytes long. The last element is a sequence (16) that is 20 bytes long. The following 20 bytes will encode the value of the field `front` in a similar way. Most operating systems provide a library that will transform this encoding into a tree data structure. The application must then assign meaning to the nodes of the tree. For example, that the first integer leaf of the root node is the field number.

Similarly, the primary encoding used by CORBA and Real-Time Publish-Subscribe (RTPS) protocol is the CDR encoding. Primitive types are encoded sequentially with predefined sizes (i.e. number of bytes), and variable length sizes are preceded by a length. The values are preceded by a byte that contains flags that specify the byte order of values (i.e. endianness).

2.2. Binary Parsing

Binary parsing is significantly different from text-based parsing. In text-based parsing, a lexer is usually used to preprocess the text into tokens. This allows the parser to operate on higher level data and so focus on determining the structure of the data. In binary parsing, bits are usually grouped into bytes, but it is generally not possible to group the bytes into tokens (such as numbers or strings) without knowledge of the context of the bytes, which is determined by the parser.

Also, binary protocols often employ semantic strategies for efficient data representation, which prevents syntactic and semantic analysis from being decoupled. For example, the length of a byte sequence or the number of elements in a list is often specified by the value of an earlier field, such as the length field in DER.

2.3. Parsing Nomenclature

Formally speaking, a parser is software that checks syntax and determines structure of input data by constructing a data structure (parse tree). A parser generator is software that accepts a specification of a protocol, sometimes called a grammar, as input and produces a parser. A parsing framework generally consists of a parser generator and related tools, and is colloquially referred to simply as a parser despite the name clash.

Some parsing frameworks accept a grammar and parse the protocol directly rather than producing a parser. This is analogous to how a programming language interpreter executes source code directly instead of first compiling it.

2.4. Fuzzing as a Motivation

Fuzzing is the practice of automatically generating test inputs for a system under test (SUT) to attempt to find bugs[16]. The earliest fuzzers were simple generational fuzzers; they generate random sequences of bytes with no knowledge of the protocol. Unfortunately, these "dumb fuzzers" are not very effective at finding vulnerabilities because the vast majority of generated inputs are syntactically incorrect, resulting in intelligent SUTs easily rejecting the inputs[16].

Since then, sophisticated mutational and generational fuzzers have been developed and have been very successful in uncovering vulnerabilities[13]. Mutation fuzzers use a pool of inputs, called a corpus, as seeds for the fuzzing process. The most advanced fuzzers today use code coverage metrics to guide the fuzzing process towards inputs that cause the SUT to execute a larger number of lines of code.

Unfortunately, coverage metrics are not easily available for black-box systems, such as software obtained from third parties (e.g. government procurement of military software). Effective black-box fuzzers can use a grammar to specify the structure of the inputs. Grammars can be used to generate new data, or they can be used to isolate promising elements of existing data for mutation.

Thus the effectiveness of fuzzing depends on the parsing techniques available. Some parsing techniques and/or DSLs impose restrictions on the form of the grammar, often to improve performance. In contrast to other parsing frameworks use cases, the generated parsers' performance is not critical, but grammar readability, expressive power, and flexibility is critical.

2.5. DNS

The Domain Name Service is a set of linked servers that translate domain names such as www.queensu.ca to internet protocol addresses. When a given device wants to communicate with another device that it does not have direct knowledge of, it sends a request to a designated name server, which returns the address of the requested device. In this case, www.queensu.ca resolves to the IPv4 address 13.107.246.10. In order to satisfy the request, the

16	32	2	1	19	4	5	U	N	I	O	N	16	20	..DOOR..

Figure 1: ASN.1 BER Encoding

Table 1: DNS Message

Field Name	**Size**	**Type**
Transaction id	2 Bytes	Integer
Flags	2 Bytes	Bit Field
# Queries	2 Bytes	Integer
# Answers	2 Bytes	Integer
# Authority Resources	2 Bytes	Integer
# Additional Resources	2 Bytes	Integer
Queries	Variable	Seq. of Query
Answers	Variable	Seq. of Resource
Authority Res	Variable	Seq. of Resource
Additional Res	Variable	Seq. of Resource

server may in turn forward the request to a different server. The DNS protocol is used to make the requests and receive responses, which are messages, also called Protocol Data Units (PDUs). It is also used to communicate between servers.

There is only one format for the top level of a DNS message. It consists of 10 fields as shown in Table 1.

The first six fields give a transaction id (used to link requests and responses), flags (message details), and four fields that give the lengths of each of the remaining 4 fields.

The first of the four sequence fields is the queries field. This gives a list of queries that are either being asked (first bit of flags is a zero) or answered (first bit of flags is a 1). Each query consists of three fields: the name that is being asked/answered, the type of query (e.g. address, name of mail host), and the class of the message (e.g. internet).

The last three sequence fields each contain sequences of resource records that represent the answers to the queries, the identity of the servers that provided the answers, and any additional resources that may be part of the message.

The first three fields of a resource record structure are the same as the query structure. In addition, it has a time to live field, which specifies how long the data in the resource record is valid, and a length field which gives the length of supplemental data fields. The format of the supplemental data fields depends on the type of the resource record which is given in the second (type) field.

One characteristic that separates DNS from many other binary protocols is that the field that specifies the type of resource record does not occur at the beginning of the resource record. Furthermore, the first field (i.e. name) is a variable size, so that the type field is not at a constant offset from the beginning of the resource record.

The DNS encoding of names is also interesting. The specification recognizes that the format of names is a sequence of strings separated by dot characters (.), and that the same string may occur more than once in a message. Thus a Domain Name is a list of words, where the dot character is implicit. Each of the words is represented as a length byte, followed by the appropriate number of characters. However, the first 2 bits of the length byte must be zeros, limiting each word to 63 characters. The end of the sequence of words is a word that consists only of the length byte with a value of 0. If a name or suffix of a name occurs earlier in the message (for example the same top-level domain may be both in a query structure and answer resource record structure), then the first 2 bits of the length field value are both 1 and the remaining bits of the length field together with the following byte contain the offset to the beginning of a sequence of words that is included in the sequence by reference[4]. Thus the entire string in a query can be copied to an answer record using two bytes. We were unable to find any offsets that made use of bits in the length field, so only the bits in the following byte are used as an offset in the grammars developed for this paper.

For these reasons, DNS provides an interesting example to use to compare the various binary parsing frameworks.

2.6. Binary Parser Generator Frameworks

There are many binary parsing frameworks in use today [17]. ANTLR is a very popular parser generator that is designed and used for parsing Domain Specific Languages (DSLs). Kaitai Struct is a parsing framework designed for binary data. Construct is a related project that is specific to the Python programming language and does not require a compilation step[18]. FormatFuzzer is another parser generator that was developed specifically for fuzzing. Spicy was built to parse network packets for the purpose of analyzing them inside of an Intrusion Detection System (IDS). SCL is based on ASN.1, and has been used for fuzzing[19] and in an IDS[20]. Full disclosure: SCL is a project in our research group. These parsers were selected for their diversity, popularity, and/or apparent potential for use in a fuzzing system.

2.7. LangSec Platform

Similar work has been done to compare parsers for security purposes. The LangSec Platform has been developed to evaluate binary parser generators for the purpose of using generated parsers to protect applications from corrupt inputs[21]. However, our interest is in using parsers for the purposes of fuzzing (rather than for use in production systems) which presents different priorities and requirements.

3. Grammar Verification

During the development and testing of the DNS grammar for each framework, pcap files were used to ensure completeness and correctness.

We concatenated a file from wiki.wireshark.org with traffic we recorded to create a pcap file that contains every resource record type present in our grammars. We also used a pcapng file from packetlife.net to test how parsers handle extra bytes at the end of a packet.

4. Implementing DNS with Limited Linguistic Features

As the community of each parsing framework has different conventions and practices, we have tried to write each grammar in a way that is consistent with other grammars where possible while staying true to the way that each framework is typically used. The complete, working grammars we developed are available online[7].

4.1. Declarative vs. Imperative

The most visually obvious classifcation of the DNS grammars is that most have a declarative style while a few are imperative, which means that control flow is expressed explicitly. The Domain Name structure in the DNS protocol can be described declaratively, as in Listing 2. The grammar specifies that a `domain` *is* a array of `words`.

```
1 domain:
2   seq:
3     - id: name
4       type: word
5       repeat: until
6       repeat-until: "_.length ==
      0 or _.length >= 192"
```

Listing 2: Domain Name structure definition in Kaitai Struct

In contrast, the FormatFuzzer grammar specifies *how* to parse a `domain` by giving a sequence of steps.

```
1 typedef struct {
2   local int length = 0;
3   do {
4     Word words;
5     length++;
6   } while(words.length != 0 && words.
      length < 192);
7 } Domain;
```

Listing 3: Domain Name structure procedure in FormatFuzzer

[7]https://github.com/thomasrdean/BinaryPaperComparison

Spicy has elements of both declarative and imperative styles. Each structure in the grammar is specified by a sequence of steps, but annotations are provided to support some features. In this case, parsing a `domain` involves just one step: parsing an array of `Label`. The `until` annotation specifies the condition for identifying the last element of the array.

```
1 type Domain = unit {
2   words: Word[] &until=($$.len == 0 ||
      $$.len >= 192);
3 };
```

Listing 4: Domain Name structure in Spicy

4.2. Factoring Grammars

In formal language theory, Backus-Naur form (BNF) grammars use the | connective between alternative productions. The DNS protocol specification[4] defines several possible formats for resource records. This concept can easily be represented using the | connective in SCL, shown in Listing 5.

```
1 ResourceRecord ::= (
2   ResourceRecordA |
3   ResourceRecordNS |
4   ResourceRecordCNAME |
5   ...
6 )
```

Listing 5: Unfactored Grammar in SCL

Some of the simplest parsers in computer science literature are top-down parsers called LL(1) parsers[22]. They begin with the start non-terminal and choose which production to use by inspecting the next token in the input stream. As mentioned in Section 2.5, several (in fact all) types of resource records begin with the same fields. Consequently, an LL(1) parser cannot determine which type of resource record it is parsing until it begins parsing it. Therefore, an LL(1) grammar must be factored to include these fields in a resource record header.

The reason that SCL can handle an un-factored grammar is that it is more powerful than LL(1) parsers. It attempts to parse each alternative and backtracks if it encounters an error. In contrast, some binary parsers use a strategy that is even simpler than that of LL(1) parsers.

Pure BNF grammars that are LL(0) are not interesting because they make decisions based on the next 0 tokens, and can therefore only parse a fixed sequence of tokens. In the case of binary parsers, BNF grammars have been extended with semantic mechanisms because binary data is context dependent (so it cannot generally be parsed using a pure BNF grammar). Kaitai Struct and FormatFuzzer both produce LL(0) parsers that rely on these mechanisms to make all control flow decisions. There is no | connective

in their grammars; the grammars explicitly state how the parser should choose how to parse the next bytes.

Listing 6 shows how the decision between different types of resource records is written in Kaitai Struct. The `name` and `type` fields have been factored out into a header, and the `switch-on` and `cases` YAML keys tell state how to choose between parsing an `rr_body_a` or another type of resource record body.

```
1 resource_record:
2 seq:
3   - id: name
4     type: domain
5   - id: type
6     type: u2
7     enum: rr_type
8   - id: body
9     type:
10    switch-on: type
11    cases:
12      "rr_type::a": rr_body_a
13      "rr_type::ns": rr_body_ns
14      ...
15 ...
16 enums:
17   rr_type:
18     1: a
19     2: ns
20     ...
```

Listing 6: Factored Grammar in Kaitai Struct

Since the length of the header is not known until it has been parsed, an LL(k) grammar will also need to use the factored form.

It may seem that ANTLR's Adaptive LL(*) parsing algorithm can predict the resource record type before parsing it. Unfortunately, as discussed in Section 5.1, the semantic constraints restrict the use of the adaptive parsing algorithm.

4.3. Iteration

Since the primary motivation for using binary protocols over text-based protocols is efficiency, it is not surprising that binary protocols regularly incorporate patterns to represent data in a compressed format. As described in Section 2.5, the top level DNS protocol structure contains several fields that communicate the number of elements in the sets of query and resource record structures which follow. This concept cannot be expressed using a BNF grammar alone and is expressed differently in different parsing frameworks. SCL grammars resemble BNF, but extra features are supported in the form of constraints (see Listing 7). The `question` field is a set of `Query` structures (defined on line 5) and the length of the set is constrained to be the value of the `numQuestion` field on line 10.

```
1 PDU ::= SEQUENCE {
2   ...
3   numQuestion INTEGER (SIZE 2 BYTES),
4   ...
5   question    SET OF Query (SIZE
      CONSTRAINED),
6   ...
7 } (ENCODED BY CUSTOM)
8 <transfer>
9   ...
10  Forward { CARDINALITY(question) ==
      numQuestion }
11  ...
12 </transfer>
```

Listing 7: Iteration in SCL

ANTLR also uses a BNF-like metasyntax, but instead extends the grammar with regular expression operators as well as ways to inject code from the grammar into the parser. This is further discussed in Section 5.1.

Construct overrides Python's indexing operation to create syntactic sugar for a higher-order function. This is further discussed in Section 5.3.

```
1 dns = Struct(
2   ...
3   "question_count" / Int16ub,
4   ...
5   "questions" / query_record[this.
      question_count],
6   ...
7 )
```

Listing 8: Iteration in Construct

Spicy and Kaitai Struct both have special annotations. FormatFuzzer is able to express iteration very naturally due to its imperative nature. These are relatively simple and were included in the listings in Section 4.1.

4.4. Constraints

Parsing an input unambiguously requires sufficient constraints on how the input is interpreted. However, many binary protocols, including DNS, provide more constraints than strictly necessary.

Each resource record contains a field, `dataLength`, that specifies the length of the rest of the resource record. Additionally, the length of that data can usually be determined by parsing it. This redundancy is very helpful for parsers that don't need to determine the structure of the remaining data; they can simply skip the specified number of bytes, or parse them into a binary blob. For the purposes of fuzzing, the structure of this data is important, so the grammars developed for this paper generally rely on the official structure of each resource record type.

However, the RRSIG resource record contains a cryptographic signature whose length is not easy to determine.

It depends on which type of algorithm is being used, and some algorithm types are reserved for future or for private use[23]. The grammars developed for this paper instead rely on the `dataLength` field of the resource record when parsing an RRSIG resource record. This is a bit awkward because the length of a Domain Name is included in `dataLength`. Some frameworks can compute the length of the Domain Name during parsing so that it can be subtracted from `dataLength`. In others, the length of a helper structure, which contains the Domain Name and the signature, is set to `dataLength`.

A language capable of expressing both constraints in these cases could be useful for a fuzzing engine because both constraints could be used or falsely assumed by the SUT. All the frameworks can enforce redundant constraints in different ways, but it may be easier for a fuzzing engine to make use of those that are written declaratively.

4.5. End Of Stream

In each of the frameworks, the procedure for parsing grammatical structures does not enforce that the entire input stream must be consumed for parsing to succeed. Each structure is parsed and the remaining bytes are left for subsequent structures to consume. Otherwise, we would not be able to parse several structures in a sequence. We call this lenient behaviour because the procedure allows extra bytes to be present.

However, the top level grammatical structure is special in that it represents the entire input, so there is no subsequent structure that may use any remaining bytes. Since DNS packets generally do not contain extra bytes, and lenient behaviour could cause issues if the parser is used to identify DNS packets, we have chosen to make the parsers strict.

Of all the frameworks, only FormatFuzzer is strict by default. Construct, ANTLR, and SCL all have keywords to specify strict behaviour. A constraint written in an annotation can be used in Spicy. A more creative solution is used for Kaitai Struct, where the input stream is examined, and, if it has not all been consumed, a failure is triggered by intentionally reading past the end of the input stream.

4.6. Hoisting

Construct and FormatFuzzer both force grammars to specify structures before referencing them. For example, the `Domain` structure must be defined earlier in the file than the `ResourceRecord` structure because `ResourceRecord` references `Domain`. In the case of DNS, this is merely inconvenient because it makes forces the structures to be in a bottom up order in the grammar. If the protocol has recursion between structures, then this is a problem. Consider a new DNS-like protocol in which a `Domain` could contain a `ResourceRecord`. Without hoisting, it is not possible to write a grammar for the new protocol because there is no ordering of structures that will avoid referencing a structure before defining it.

5. Comparison of Frameworks

After discussing how several features in the DNS protocol are encoded in grammars for different frameworks, we now focus on considering each framework.

5.1. ANTLRv4

ANTLR grammars are written in the ANTLR DSL and are typically used in turn to specify DSLs. The Adaptive LL(*) or ALL(*) parsing algorithm is quite advanced and is highly optimized for parsing DSLs. As the documentation indicates, it is possible to use ANTLR on binary data[24]. This is done by converting each byte in the input stream into a character in the input stream for the lexer which in turn feeds its output to the parser. (The separation between the lexer and parser is useful in text-based parsing because it allows the lexer to group up characters into tokens such as keywords, numbers, and strings before the parser constructs the abstract syntax tree.)

In practice however, the differences between text-based and binary parsing go beyond swapping characters and bytes because binary protocols are often context dependent and non-ambiguous, as mentioned in Section 2.2. We found that the complex problems relating to ambiguity that ANTLR addresses in text-based parsing are different from the concepts that regularly appear in binary protocols. This is clearly illustrated in the DNS protocol where the parser must decide between resource record types.

This is implemented as a set of parser rules in Listing 9.

```
1 resourceRecord:
2   name=domain
3   body=rrBody
4   ;
5 rrBody
6   : resourceRecordA
7   | resourceRecordNS
8   ...
9   ;
10 resourceRecordA:
11   type_=typeA
12   class=uint16
13   timeToLive=uint32
14   dataLength=uint16
15   address=ipv4Address
16   ;
17 resourceRecordNS:
18   type_=typeNS
19   class=uint16
20   timeToLive=uint32
21   dataLength=uint16
22   nameServer=domain
23   ;
24 ...
```

Listing 9: Resource Records in ANTLR

As mentioned in Section 4.2, this grammar is factored, but in a slightly different way than the LL(0) grammars. Rather than reading in the `type_` field and writing imperative control flow code that makes the decision between different resource record types based on its value (making use of semantic information), a declarative style is used and the decision is made just before the `type_` field is consumed. The ALL(*) parsing algorithm compares the next 2 tokens (which form the `type_` field) to the types of tokens expected by each of the `resourceRecord*` rules and picks the matching rule. Note that the ALL(*) parsing algorithm is capable of complex analysis, but a simple choice based on the next 2 bytes is sufficient in this case. (It may seem as though this could be an LL(1) decision, but it is LL(2) because the lexer is not capable of grouping the 2 bytes together.)

For this to work, the semantic information (the value of the `type_` field) must be converted into syntactic information (the type of the tokens in the field). This is done by a combination of parser rules and also lexer rules, whose names are in all capitals, in Listing 10. Note that ANTLR converts all bytes to UTF 16 on input, so all byte values are given as 16 bit values in the grammar.

```
1  typeA: '\u0000' data=TYPE_A ;
2  typeNS: '\u0000' data=TYPE_NS;
3  ...
4  byte returns [uint8_t val]
5    : data=allTerminals
6    {
7  $val = $data.text[0];
8    };
9
10 nullByte: NULL_BYTE;
11 letterLength returns [uint8_t val]
12   : data=letterByte
13   {
14 $val = $data.text[0];
15   };
16 letterByte:
17   LETTER_LENGTH_BYTE
18   | TYPE_A
19   | TYPE_NS
20   ...
21   ;
22 refByte: REF_BYTE;
23
24 allTerminals
25   : NULL_BYTE
26
27   | TYPE_A
28   | TYPE_NS
29   ...
30
31   | LETTER_LENGTH_BYTE
32   | REF_BYTE
33
34   | BYTE
35   ;
36
37 NULL_BYTE: '\u0000';
38
39 TYPE_A: '\u0001';
40 TYPE_NS: '\u0002';
41 ...
42
43 LETTER_LENGTH_BYTE: '\u0001' .. '\u0040
       ';
44 REF_BYTE: '\u00c0'..'\u00ff';
45
46 BYTE: '\u0000'..'\u00ff';
```

Listing 10: Lexer and Related Parser Rules in ANTLR

The ANTLR lexer creates a token for each byte in the input stream. Each token has a type, named after the lexer rule selected by the lexer. The lexer runs without any knowledge of the parser, so it cannot classify bytes into tokens based on which tokens the parser expects. It chooses the first lexer rule whose range of values contains the byte's value.

This means that the order the lexer rules appear in the grammar matters. For example, a byte with value 1 is classified as a `TYPE_A`, even though it also fits under the range of values for a `LETTER_LENGTH_BYTE` and a `BYTE` because `TYPE_A` is defined earlier in the grammar than the others.

We could have created a token type for all 256 possible byte values, but we combine those that the parser doesn't need to be able to differentiate. In some cases, different grammar rules have different ranges of byte values that select alternatives, and the different ranges overlap. For example, the 0x01 byte is used both as a value for the length at the the start of a domain name word and as the value of the type for an IPv4 address resource record. The lexer can only classify the value as one token type which is `TYPE_A`. To allow the 0x01 byte to be used as the length at the start of a domain name word, we use the non-terminal `letterByte` to combine the otherwise distinct tokens.

The `{$val = ...}` lines are actions that work to turn the input characters into a numerical values that can be used by semantic predicates and (other) actions, which are mechanisms that allow the grammar to inject code into the parser. This code is written in between curly brackets in the language of the host application, which is C++ in this case. The `$` symbol is replaced by code that references the current parsing context. The code is then executed at parse-time and can affect parser control flow.

As mentioned in Section 4.3, the general iteration concept in the DNS protocol cannot be defined by a BNF grammar alone. The implementation in Listing 11 is obtuse, but is the most reasonable way we could encode iteration in ANTLR. Lines 6 and 7 show how the values of `numQuestion` and `numAnswer` are passed as parameters to the parser rules that carry out the iteration. These rules each contain 2 semantic predicates that control parser flow. To properly constrain the iteration, we need to prevent too

many elements (`$i < $n`) and separately prevent too few (`$i == $n`). The asterisk is a regular expression symbol that indicates that 0 or more of the preceding items may be present and the parentheses group several items into one. `$i++;` is an action that causes the whole rule to function similarly to a for loop in the C programming language.

Unfortunately, the concept of iteration must be specified for each type of element because we could not find a reasonable way to make the iteration code generic. This results in duplicate code (`setOfQuery` and `setOfResourceRecord`).

```
1 dns:
2   ...
3   numQuestion=uint16
4   numAnswer=uint16
5   ...
6   question=setOfQuery[$numQuestion.val]
7   answer=setOfResourceRecord[$numAnswer
      .val]
8   ...
9   ;
10
11 setOfQuery [int n]
12 locals [int i = 0]
13   : ( {$i < $n}? query {$i++;} ) * {$i
       == $n}?
14   ;
15 ...
16 setOfResourceRecord [int n]
17 locals [int i = 0]
18   : ( {$i < $n}? resourceRecord {$i++;}
       ) * {$i == $n}?
19   ;
```

Listing 11: Iteration in ANTLR

As mentioned in Section 4.2, we were unable to get ANTLR to behave correctly with an unfactored grammar. We tried making a small change to the ANTLR DNS grammar described above, moving the `name=domain` field from the `resourceRecord` rule to each of the `resourceRecord*` rules. The result is shown in Listing 12.

```
1 resourceRecord
2   : resourceRecordA
3   | resourceRecordNS
4   ...
5   ;
6 resourceRecordA:
7   name=domain
8   type_=typeA
9   class=uint16
10  ...
11  ;
12 resourceRecordNS:
13  name=domain
14  type_=typeNS
15  class=uint16
16  ...
17  ;
```

Listing 12: Dysfunctional Resource Records in ANTLR

It may seem that this change has no effect on the behaviour of the parser. However, this actually causes many packets to be parsed incorrectly.

The ALL(*) prediction mechanism never backtracks. Instead, the idea is to "launch subparsers at a decision point, one per alternative production"[12]. The subparsers simulate how the input would be parsed if each production is chosen to see if there are any valid paths stemming from the production in question. This prediction mechanism ignores the semantic predicates in the DNS grammar when it has to choose a resource record rule. This is because the semantic predicates reference variables such as `i` that are not defined yet[25]. (ANTLR calls these "context-dependent" semantic predicates.) Consequently, the prediction mechanism sees a grammar that is considerably under-constrained. For example, any number of `words` are allowed in `domains` and any number of `resourceRecords` can be in each of the `answer`, `authority`, and `additional` fields.

Using ANTLR's debugging features, we have observed that the prediction mechanism is often unable to identify which resource record rule to use. A warning that the grammar is ambiguous is emitted when the parser is in the `resourceRecord` rule, indicating that several resource record rules appear to be valid. When there are several valid productions, ANTLR resolves the ambiguity according to the order of the productions in the rule[12]. In this case, `resourceRecordA` has priority over the others. After the warning about ambiguity, we observe that the parser incorrectly chooses `resourceRecordA`, and then we see an error that a token failed to match `'\u0001'` because the actual `type_` in the input data is not `TYPE_A`. To ensure we understood the cause of ANTLR's behaviour, we developed a minimum working example.

Making this change to the grammar also increases the time it takes to parse our test pcap file from under 3 seconds to over 10 minutes. It is difficult to be entirely sure of what is going on because the prediction engine is complex. When the grammar is compiled, an augmented recursive transition network (ATN) is generated to optimize the prediction mechanism. Deterministic finite automata are also constructed and cached during parsing[12]. This makes it very difficult for programmers who have not worked on the ANTLR source to understand the behaviour of the prediction mechanism[12].

5.2. Kaitai Struct

The Kaitai Struct framework translates a specification file in yaml format into a variety of forms, some of which are useful for understanding the specified protocol. Other forms include source code in Python, C++, and Java. For

this survey, the yaml was compiled into a Python class that can be used in a larger Python program.

Listing 13 shows a portion of the DNS specification expanded from the version provided by the Kaitai Struct developers and simplified in some areas to be comparable to the grammars for the other frameworks. We have provided the full grammar as part of the source code.

The specification file (.ksy) for DNS consists of four sections. The first section (meta) provides global information such as the name of the spec, default byte order (endianess) and the default encoding of strings in the binary data. This is followed by the specification of the root message type. Each field is identified with a hyphen, and the attributes of that field. The value u2 as a type indicates a 2 byte unsigned integer. The special `switch-on` and `if` tags give control flow information to the parser and the `instances` tag allows `is_ref` to be computed and later referenced. Fields that are sequences of another type have a special tag indicating how their length is constrained.

```
meta:
  id: dns_packet
  endian: be
  encoding: utf-8
seq:
  ...
  - id: qdcount
    type: u2
  ...
  - id: queries
    type: query
    repeat: expr
    repeat-expr: qdcount
  ...
types:
  ...
  resource_record:
    seq:
      - id: name
        type: domain
      - id: type
        type: u2
        enum: rr_type
      - id: body
        type:
          switch-on: type
          cases:
            "rr_type::a": rr_body_a
            "rr_type::ns": rr_body_ns
  ...
  domain:
    seq:
      - id: name
        type: word
        repeat: until
        repeat-until: "_.length == 0 or
    _.length >= 192"
  word:
    seq:
      - id: length
        type: u1
      - id: ref
        if: "is_ref"
        type: u1
      - id: letters
        if: "is_letters"
        type: str
        size: length
      - id: check
        type: fail
        if: "not is_ref and not
    is_letters"
    instances:
      is_ref:
        value: length >= 192
      is_letters:
        value: length < 64
  ...
  fail:
    seq:
      - id: eat_bytes
        size-eos: true
      - id: fail_to_eat_another
        size: 1
  ...
enums:
  rr_type:
    1: a
    2: ns
    ...
```

Listing 13: Kaitai Struct Grammar for DNS

Kaitai Struct is a large project with many specialized tags for specific purposes, but at its core is LL(0). Unlike any of the other frameworks in this paper, it does not support serialization (entirely), which would be important if it were to be integrated with a fuzzing system.

5.3. Construct

Construct differs from all the other frameworks in this survey in that a Construct grammar is code that references the Construct library rather than a DSL in a file that is read by the framework. Construct does have an experimental compilation option, but typically the grammar code parses the protocol directly (see Section 2.3) [26]. The library's intended use is to serialize and deserialize Python objects in a larger Python program so that the objects can be saved to disk or sent over a network.

```
word = Select(
  "ref" / Struct(
    "first_byte" / Int8ub,
    Check(this.first_byte >= 0xc0),
    "ref" / Int8ub,
  ),
  "label" / Struct(
    "length" / Int8ub,
```

```
9       Check(this.length < 0x40),
10      "letters" / Byte[this.length],
11    ),
12 )
13
14 domain = RepeatUntil(lambda x,lst,cts:
       hasattr(x, "ref") or x.length == 0,
       word)
15
16 query_record = Struct(
17   "name" / domain,
18   "type" / Int16ub,
19   "class" / Int16ub,
20 )
21 ...
22 rrA = Struct(
23   "name" / domain,
24   "type" / Int16ub,
25   Check(this.type == 0x01),
26   "class" / Int16ub,
27   "timeToLive" / Int32ub,
28   "dataLength" / Int16ub,
29   "address" / ipv4Address,
30 )
31 ...
32 resource_record = Select(
33   rrA,
34   rrNS,
35   ...
36 )
37
38 dns = Struct(
39   "id" / Int16ub,
40   "flags" / Int16ub,
41   "question_count" / Int16ub,
42   ...
43   "questions" / query_record[this.
       question_count],
44   ...
45 )
```

Listing 14: Construct Grammar for DNS

The grammar, shown in Listing 14, is adapted from a deprecated example[27] in the project repository. `Select` is a higher-order function that implements back-tracking which is triggered by `Check`. `this` is a global singleton object that returns lambdas when its properties are accessed. The `==` and indexing operators have been overloaded to operate on lambdas by returning composed lambdas. The code that supports these features is inside the Construct library, but they could alternatively be defined inside the grammar. This means that the framework is easily extensible by writing additional code which extends Construct right inside the grammar.

An imperative style can also be used to express the desicion between resource record types. In Listing 15, the grammar has been factored to be LL(0) and the `Select` constructor (function) has been replaced with `Switch`.

```
1 rrA = Struct(
2   "class" / Int16ub,
3   "timeToLive" / Int32ub,
4   "dataLength" / Int16ub,
5   "address" / ipv4Address,
6 )
7 rrNS = Struct(
8   "class" / Int16ub,
9   "timeToLive" / Int32ub,
10  "dataLength" / Int16ub,
11  "nameServer" / domain,
12 )
13 ...
14 resource_record = Struct(
15   "name" / domain,
16   "type" / Int16ub,
17   Switch(this.type, {
18     1: rrA,
19     2: rrNS,
20     ...
21   })
22 )
```

Listing 15: Constuct Grammar using Switch

Debugging the deprecated grammar made it very obvious that higher-order functions and operator overloading can be disorienting. On the other hand, the absence of a compilation step makes for a very smooth development experience, especially because the grammar and the library can all be loaded into a debugging environment at the same time.

As discussed in Section 4.6, the lack of support for hoisting forces the grammar structures to be ordered with lower level structures appearing first.

5.4. FormatFuzzer

FormatFuzzer is a framework that can parse and generate binary data according to a grammar, which they refer to as a binary template[28]. The binary templates are an extension those used by the 010 Editor[13] and have "a similar syntax to C/C++ structs but they are run as a program. Every time a variable is declared in the template, the variable is mapped to bytes in the current file"[29]. In Listing 16, the variable transactionId at line 35 is mapped to the first 2 bytes (16 bits) of the PDU. The resulting compiled C code is readable and easier to debug than the code generated by ANTLR.

Line 1 calls a function defined in the FormatFuzzer library to configure the parser for big-endian data. The framework also provides other built-in functions that support inspecting and even searching through the input stream.

```
1 BigEndian();
2 ...
3 typedef struct {
4   ubyte length;
5   if(length != 0) {
```

```
6     if(length < (ubyte) 64) {
7       ubyte letters[length];
8     } else if (length >= (ubyte) 192) {
9       ubyte reference;
10    } else {
11      Exit(1);
12    }
13  }
14 } Word;
15
16 typedef struct {
17   local int length = 0;
18   do {
19     Word words;
20     length++;
21   } while(words.length != 0 && words.
       length < 192);
22 } Domain;
23 ...
24 typedef struct {
25   Domain name;
26   uint16 type_;
27   switch(type_) {
28     case 1: RRBodyA body; break;
29     case 2: RRBodyNS body; break;
30     ...
31   }
32 } ResourceRecord;
33
34 struct DNS {
35   uint16 transactionId;
36   uint16 flags;
37   uint16 numQuestion;
38   ...
39   Query question[numQuestion];
40   ...
41 } file;
```

Listing 16: FormatFuzzer Binary Template for DNS

This binary template was written from scratch in accordance with examples from the 010 Editor template repository.

As discussed in Sections 4.2 and 4.3, the imperative style of FormatFuzzer forces the grammar to be LL(0), but supports iteration very naturally with `while` and `do-while` loops. This allows FormatFuzzer to produce very fast parsers. However, it is not obvious that the trade-off of gaining performance while losing expressive power is justified for fuzzing systems, which generally run off-line.

FormatFuzzer has several novel features that are useful for fuzzing. It is capable of noting each decision that is made as an input is parsed and recording these decisions to a file. The file can be mutated and then used to generate another input that is different, but related to the original. FormatFuzzer can also mine magic values from grammars for future use in fuzzing[13].

5.5. Spicy

Spicy is a framework including a parser generator that compiles the Spicy DSL to HILTI, an intermediary language, which is then compiled to C++ code. Parsers generated by Spicy are specially designed to operate as protocol analyzers for the Zeek network traffic analysis platform (or IDS), but can also be linked into other projects[30]. Spicy generates "generates a non-backtracking recursive-descent LL(1) parser, in the form of HILTI code closely following the production structure"[31].

The grammar in Listing 17 is adapted from the DNS grammar in one of Zeek's repositories.

```
1  public type PDU = unit {
2    transactionId     : uint16;
3    flags : bytes &size=2;
4    numQuestion: uint16;
5    ...
6    question:    Query[self.numQuestion];
7    ...
8  };
9  ...
10 type ResourceRecord = unit() {
11   name:    Domain;
12   type_:   uint16 &convert=RRType($$);
13   switch ( self.type_ ) {
14     RRType::A     -> bodyA: RRBodyA;
15     RRType::NS    -> bodyNS: RRBodyNS;
16     ...
17   };
18 };
19
20 type RRType = enum {
21   A = 1,
22   NS = 2,
23   ...
24 };
25
26 type RRBodyA = unit() {
27   class_: uint16;
28   timeToLive: uint32;
29   dataLength: uint16;
30   address: addr &ipv4;
31 };
32 type RRBodyNS = unit() {
33   class_: uint16;
34   timeToLive: uint32;
35   dataLength: uint16;
36   nameServer: Domain;
37 };
38 ...
39 type Domain = unit {
40   words: Word[] &until=($$.len == 0 ||
       $$.len >= 192);
41 };
42
43 type Word = unit() {
44   len: uint8 &requires=($$ < 64 || $$
       >= 192);
```

```
45   reference: uint8 if (self.len >= 192)
       ;
46   letters: bytes &size=self.len if (
       self.len < 64);
47 };
```

Listing 17: Spicy DNS Grammar

The decision between resource record types is written imperatively on line 13, which chooses a unit (an ordered set of fields) based on the value of `type_`, which has already been parsed.

Alternatively, this can be written declaratively, as seen in Listing 18, similarly to the ANTLR grammar. This is because Spicy (unlike Kaitai Struct and FormatFuzzer) is able to handle grammars with implicit decisions.

```
1 type ResourceRecord = unit() {
2   name:   Domain;
3   switch {
4     -> bodyA: RRBodyA;
5     -> bodyNS: RRBodyNS;
6     ...
7   };
8 };
9
10 type RRBodyA = unit() {
11   type_: uint16(1);
12   class_: uint16;
13   ...
14 };
15 type RRBodyNS = unit() {
16   type_: uint16(2);
17   class_: uint16;
18   ...
19 };
```

Listing 18: Spicy DNS Grammar Utilizing Lookahead

When the parser reaches the `switch` statement on line 3, it must first tokenize 2 bytes into a `uint16` (like an LR(1) parser). Then, it predicts which resource record body unit follows (like an LL(1) parser). If the grammar is changed so that several `RRBody*` units begin with fields that can't be distinguished, the grammar will not compile.

Unlike the other parsers, Spicy supports explicit backtracking [32], which isn't exercised by this DNS grammar.

Spicy has an additional feature that may be beneficial for a hypothetical fuzzing system; it supports casting data to enumerations while retaining the original data. In the grammar, the `type_` field of `ResourceRecord` is cast to `RRType` on line 12 of Listing 17, but the original bits are preserved in `$$`.

5.6. SCL

SCL is a DSL and parser generator based on ASN.1, so SCL grammars are very similar in style to how protocols are naturally specified (without refactoring to appease a parser generator)[19]. The grammars are converted into C code using TXL, a special-purpose programming language[33].

It is a recursive descent parser with backtracking and static LL(k) optimization. This means that it tries to find a static lookahead distance for each decision at compile time and decides to use backtracking if it can't. The LL(k) optimization allows grammars to use the `|` connective without incurring the cost of backtracking in most cases.

A unique feature of SCL is seen in Listing 19: there are explicit constraints. This provides a dividing point between syntactic constraints (in the parts of the grammar that have a BNF style) and the semantic constraints (which are written between the "transfer" braces that look like xml tags). Additionally, explicit constraints may be easier for a fuzzing system to process, which could help it to produce inputs that test how the SUT handles each constraint.

```
1 DNS DEFINITIONS ::= BEGIN
2   EXPORTS PDU;
3
4   PDU ::= SEQUENCE {
5     transactionId   INTEGER (SIZE 2
      BYTES),
6     flags   OCTET STRING (SIZE 2 BYTES)
      ,
7     numQuestion INTEGER (SIZE 2 BYTES),
8     ...
9     question    SET OF Query (SIZE
      CONSTRAINED),
10    ...
11  } (ENCODED BY CUSTOM)
12  <transfer>
13    ...
14    Forward { CARDINALITY(question) ==
      numQuestion }
15    ...
16    All Bytes Used
17  </transfer>
18
19  Query ::= SEQUENCE {
20    name  Domain (SIZE DEFINED),
21    type  INTEGER (SIZE 2 BYTES),
22    class INTEGER (SIZE 2 BYTES)
23  }
24
25  ResourceRecord ::= (
26    ResourceRecordA |
27    ResourceRecordAAAA |
28    ...
29  )
30
31  Domain ::= SEQUENCE {
32    words SET OF Word (SIZE CONSTRAINED
      )
33  }
34  <transfer>
```

```
35      Forward { TERMINATE(words) ==
        EndWord }
36    </transfer>
37
38
39    Word ::= (InlineWord | EndWord)
40    EndWord ::= (ReferenceWord | NullWord
        )
41
42    InlineWord ::= SEQUENCE {
43      length  INTEGER (SIZE 1 BYTES),
44      letters OCTET STRING (SIZE
        CONSTRAINED)
45    }
46    <transfer>
47      Forward { LENGTH(letters) == length
         }
48      Back { length != 0 }
49      Back { length != 192 && length !=
        193 } -- 192 is c0
50    </transfer>
51
52    ReferenceWord ::= SEQUENCE {
53      header INTEGER (SIZE 1 BYTES),
54      reference INTEGER (SIZE 1 BYTES)
55    }
56    <transfer>
57      Back { header == 192 || header ==
        193 }
58    </transfer>
59
60    NullWord ::= SEQUENCE {
61      header INTEGER (SIZE 1 BYTES)
62    }
63    <transfer>
64      Back { header == 0 }
65    </transfer>
66  END
```

Listing 19: SCL Grammar for DNS

The `Forward` constraints are used to limit the potential parsing options. For example, the parser knows that the size of `question` is the value it parsed for `numQuestion` before it begins to parse `question`. `Back` constraints trigger backtracking (if the LL(k) optimization failed during compilation). This allows `Back` constraints to reference the content of the structure they correspond to, as in the `transfer` blocks for `ReferenceWord` and `NullWord`.

Representing some constraints declaratively is more complex. Notice that the `EndWord` nonterminal does not seem to be necessary. It was introduced (and the production for `Word` was split) because the constraint on `Domain` uses the `TERMINATE` feature, which does not work with the `||` connective. The constraint engine could be extended to make this work, but it is not clear that writing the `||` connective in the constraint is more articulate than introducing the `EndWord` nonterminal.

Table 2: Summary of Framework Features

Framework	Parsing Algorithm	Serialization	Existing Grammars	Host Languages	Compiled
ANTLR	ALL(*)	N	😵	many	Y
Kaitai Struct	LL(0)	😵	😍	many	Y
Construct	Backtracking	Y	😐	Python	😵
FormatFuzzer	LL(0)	Y	😍	C++	Y
Spicy	LL(1)	N	😍	C++	Y
SCL	Backtracking	Y	😐	C	Y

6. Summary

We have collagated some details and subjective observations about each of the frameworks in Table 2. The expressive power of each framework influences how grammars are written. However, implementing concepts of binary protocols often relies on using special features of the framework. These include semantic predicates in ANTLR, special YAML keys in Kaitai Struct, built-in functions in FormatFuzzer, annotations in Spicy, and special features in constraints in SCL. Since most frameworks provide ways to inspect the input stream using special features, an LL(k) grammar can generally be used, but it may be difficult to write, debug, and understand.

For each parser, we also show whether or not it can perform serialization, the comprehensiveness of the existing repository of grammars available online, which programming languages the resulting parser can be easily integrated with, and whether or not the grammar is compiled into a parser at all.

Kaitai Struct has some support for serialization, but it does not cover the entire feature set of the language. Construct has an experimental compilation feature, which does not work with some language features. ANTLR and Kaitai Struct can both output parsers in Java, C++, C#, Python, JavaScript, and other languages. Kaitai Struct grammars are completely agnostic to the output language. ANTLR grammars, on the other hand, contain semantic predicates and actions, which are written in the output language.

7. Conclusion

It is clear that each parsing framework has different strengths and weaknesses. When choosing a binary parser generator for a project, we recommend first considering concepts in the given binary protocol(s)/file(s) that require backtracking or cannot easily be implemented in a BNF-style grammar. Then, look for how similar concepts are treated in the grammars developed for this paper as well as in any grammars available online. This should give an indication as to how difficult it will be to write the

grammar(s) required for the project. If there is a existing grammar for the protocol, it can be a useful starting place even if the grammar needs to be modified heavily to suit the needs of the larger system or re-written for a different framework.

It is our opinion that attempts to implement specific concepts in binary parsing using general parser features are often awkward and should be avoided. For example, the code that expresses the concept of iteration in ANTLR using actions and semantic predicates was difficult to write and debug. Therefore, it is important to make sure that the framework has sufficient linguistic features or that they can be added easily. Adding linguistic features to the frameworks involves modifying the framework's source code, except in the case of Construct. Since Construct does not require a compilation step, it is easily extensible. The larger class of parser generators that work this way are called combinator parsers and are well-studied[34]. These may be desirable for fuzzing because fuzzing is by nature a higher-level operation. It seems that there has been some work done on combinator parser-fuzzing systems[35], but it is not developed or it is not well documented[36].

Parsing binary protocols does not generally require backtracking, but backtracking allows for more expressive grammars. If there is a specification for a protocol that is written declaratively, it will probably be easiest to write the grammar declaratively and without factoring it to work with an LL(*) parsing framework. In the case of the DNS protocol, SCL and Construct are the only frameworks whose grammars can express the protocol as it is described in RFC 1035[4] (see Section 4.2).

The main difference between the two frameworks is that SCL has been used in an intrusion detection system, so it is designed to produce very fast parsers. SCL takes 7 ms to parse our test pcap file, whereas Construct takes 211 ms. SCL is running a compiled C program, whereas Construct is running Python code, so some of this difference is likely due to the Python runtime warming up. On the same packets repeated 100 times in the same pcap file (5 MB), SCL takes 81 ms, whereas Construct takes 13 seconds.

Therefore SCL is significantly faster than Construct. On the other hand, Construct has in-depth online documentation.

Our research group has SCL grammars for a number of binary protocols, some of which have been published. Construct has a few grammars for binary files and protocols, but most are deprecated[37].

In general, Construct seems better suited for quickly developing grammars for prototypes and SCL is better for more established projects, where orderly grammar syntax, the ability to link parser code with other object files, and the performance of the parser become important.

Kaitai Struct is a good choice if the grammar is likely to be used for multiple projects that may use different programming languages and do not need to perform serialization. It has a large gallery of grammars available for binary files and network protocols, a large online community, and even a tool to visualize grammars.

FormatFuzzer and Spicy are similar in functionality except that FormatFuzzer is also capable of fuzzing and serializing data. FormatFuzzer has a large repository of grammars (binary templates) available for binary files and also partial grammars that provide syntax-highlighting for text-based files. Spicy has existing grammars for some binary protocols and a few binary files. It is clear that FormatFuzzer was developed for files and Spicy was developed for protocols. We are convinced that protocol parsers can be used to parse files and vice versa so long as interactions between data in different messages in a protocol are not relevant to parsing. This is demonstrated in that Spicy has a zip file grammar and a png file grammar available online and we were able to develop a DNS grammar for FormatFuzzer.

While text-based parser generators have been refined and optimized with many advanced techniques, we are convinced that it is not practical to adapt text-based parsers to operate on binary data. Therefore, we do not recommend using ANTLR to parse binary data.

References

[1] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach, T. Berners-Lee, Hypertext transfer protocol – http/1.1, RFC 2616, RFC Editor, http://www.rfc-editor.org/rfc/rfc2616.txt (June 1999).
URL http://www.rfc-editor.org/rfc/rfc2616.txt

[2] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, Extensible Markup Language (XML) 1.0 (Fifth Edition) (Nov. 2008).
URL https://www.w3.org/TR/xml/

[3] T. Bray, The javascript object notation (json) data interchange format, RFC 7159, RFC Editor, http://www.rfc-editor.org/rfc/rfc7159.txt (March 2014).
URL http://www.rfc-editor.org/rfc/rfc7159.txt

[4] P. Mockapetris, Domain names - implementation and specification, STD 13, RFC Editor, http://www.rfc-editor.org/rfc/rfc1035.txt (November 1987).
URL http://www.rfc-editor.org/rfc/rfc1035.txt

[5] D. Mauro, K. Schmidt, Essential snmp (July 2001).
URL https://doc.lagout.org/network/Essential%20SNMP%202001.pdf

[6] Common object request broker architecture (February 2021).
URL https://www.omg.org/spec/CORBA/3.4/

[7] International Telecommunication Union, Information technology – asn.1 encoding rules: Specification of basic encoding rules (ber), canonical encoding rules (cer) and distinguished encoding rules (der), ITU-T Recommendation X.690 (February 2021).

[8] OMB, Chapter 15, generalinter-orbprotocol (Jun. 2002).
URL https://www.omg.org/cgi-bin/doc?formal/02-06-51

[9] I. J. . I. Technology, Iso/iec 8211:1994 (1994).
URL https://www.iso.org/standard/20305.html

[10] Interface definition language (March 2018).
URL https://www.omg.org/spec/IDL/4.2/

[11] Wireshark, https://gitlab.com/wireshark/wireshark/-/blob/master/epan/dissectors/packet-dns.c (2022).

[12] T. Parr, S. Harwell, K. Fisher, Adaptive ll(*) parsing: The power of dynamic analysis, SIGPLAN Not. 49 (10) (2014) 579–598. doi:10.1145/2714064.2660202.
URL https://doi.org/10.1145/2714064.2660202

[13] R. Dutra, R. Gopinath, A. Zeller, Formatfuzzer: Effective fuzzing of binary file formats (2021). doi:10.48550/ARXIV.

2109.11277.
URL https://arxiv.org/abs/2109.11277
[14] R. Sommer, J. Amann, S. Hall, Spicy: A unified deep packet inspection framework for safely dissecting all your data, in: Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 558–569. doi:10.1145/2991079.2991100.
URL https://doi.org/10.1145/2991079.2991100
[15] A. ElShakankiry, T. Dean, Context sensitive and secure parser generation for deep packet inspection of binary protocols, in: 2017 15th Annual Conference on Privacy, Security and Trust (PST), 2017, pp. 77–7709. doi:10.1109/PST.2017.00019.
[16] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, C. Holler, The Fuzzing Book, CISPA Helmholtz Center for Information Security, 2021, retrieved 2021-10-26 15:30:20+02:00.
URL https://www.fuzzingbook.org/
[17] D. Loss, Awesome binary parsing, https://github.com/dloss/binary-parsing (2022).
[18] Kaitai struct: Faq (2021).
URL https://doc.kaitai.io/faq.html
[19] Y. Turcotte, O. Tal, S. Knight, T. Dean, Security vulnerabilities assessment of the x.509 protocol by syntax-based testing, in: IEEE MILCOM 2004. Military Communications Conference, 2004., Vol. 3, 2004, pp. 1572–1578 Vol. 3. doi:10.1109/MILCOM.2004.1495173.
[20] M. S. Hasan, T. Dean, F. T. Imam, F. Garcia, S. P. Leblanc, M. Zulkernine, A constraint-based intrusion detection system, in: Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems, ECBS '17, ACM, New York, NY, USA, 2017, pp. 12:1–12:10. doi:10.1145/3123779.3123812.
[21] O. Levillain, S. Naud, A. T. Rasoamanana, Work-in-Progress: Towards a Platform to Compare Binary Parser Generators, in: 35. IEEE Security and Privacy Workshops, SPW (LangSec) 2021, San Jose, CA, USA, 2021.
[22] D. Rosenkrantz, R. Stearns, Properties of deterministic top-down grammars, Information and Control 17 (3) (1970) 226–256. doi:https://doi.org/10.1016/S0019-9958(70)90446-8.
URL https://www.sciencedirect.com/science/article/pii/S0019995870904468
[23] R. Arends, R. Austein, M. Larson, D. Massey, S. Rose, Resource records for the dns security extensions, RFC 4034, RFC Editor, http://www.rfc-editor.org/rfc/rfc4034.txt (March 2005).
URL http://www.rfc-editor.org/rfc/rfc4034.txt
[24] T. Parr, Antlr v4, https://github.com/antlr/antlr4/blob/master/doc/parsing-binary-files.md (2022).
[25] T. Parr, The Definitive ANTLR 4 Reference, 2nd Edition, Pragmatic Bookshelf, 2013.
[26] Arkadiusz Bulski, Tomer Filiba, Corbin Simpson, Compilation feature (2020).
URL https://construct.readthedocs.io/en/latest/compilation.html
[27] A. Bulski, Construct 2.10, https://github.com/construct/construct/blob/master/deprecated_gallery/ipstack.py (2022).
[28] A. Zeller, Formatfuzzer, https://github.com/uds-se/FormatFuzzer (2022).
[29] Introduction to templates and scripts (2021).
URL https://www.sweetscape.com/010editor/manual/IntroTempScripts.htm
[30] Spicy — generating robust parsers for protocols & file formats (2020).
URL https://docs.zeek.org/projects/spicy/en/latest/index.html
[31] R. Sommer, J. Amann, S. Hall, Spicy: A unified deep packet inspection framework for safely dissecting all your data, in: Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 558–569. doi:10.1145/2991079.2991100.
URL https://doi.org/10.1145/2991079.2991100
[32] Parsing (2020).
URL https://docs.zeek.org/projects/spicy/en/latest/programming/parsing.html
[33] J. R. Cordy, The txl source transformation language, Sci. Comput. Program. 61 (3) (2006) 190–210. doi:10.1016/j.scico.2006.04.002.
URL https://doi.org/10.1016/j.scico.2006.04.002
[34] G. Hutton, Higher-order functions for parsing, Journal of Functional Programming 2 (09 1999). doi:10.1017/S0956796800000411.
[35] P. Anantharaman, Keggyfuzzer, https://github.com/prashantbarca/KeggyFuzzer (2017).
[36] P. Anantharaman, The langsec journey - introduction (2017).
URL http://langsec.prashant.at/
[37] A. Bulski, Construct 2.10, https://github.com/construct/construct (2022).