Investigating Automatic Code Generation for Network Packet Parsing

Stephen McQuistin
University of Glasgow, UK
sm@smcquistin.uk

Vivian Band
University of Glasgow, UK
vivianband0@gmail.com

Dejice Jacob
University of Glasgow, UK
dejice.jacob@glasgow.ac.uk

Colin Perkins
University of Glasgow, UK
csp@csperkins.org

***Abstract*—Use of formal protocol description techniques and code generation can reduce bugs in network packet parsing code. However, such techniques are themselves complex, and don't see wide adoption in the protocol standards development community, where the focus is on consensus building and human-readable specifications. We explore the utility and effectiveness of new techniques for describing protocol data, specifically designed to integrate with the standards development process, and discuss how they can be used to generate code that is safer and more trustworthy, while maintaining correctness and performance.**

I. Introduction

The code that parses incoming network packets is an important part of any protocol implementation, and problems with this code are a frequent source of security vulnerabilities [1]. Unfortunately, as a result of ambiguous and inconsistent protocol standards and specifications, typically written using informal English prose, network packet parsing code often contains logic errors and other bugs. In principle, standards documents can be made more precise by using formal protocol description techniques. This improved precision should make it more likely that the specification is correctly interpreted and implemented, and can also be used to enable automatic code generation, further improving the quality of parsing code.

In practice, formal protocol description techniques have failed to gain traction within the standards community. They often require significant changes to the engineering process by which standards are developed, and to the way specifications are written. Such changes have proven too onerous for the standards development community, and the vast majority of standards published do not make use of formal techniques.

In previous work, we have proposed structured specification techniques that *do* integrate with the standardisation process [2]. Such techniques include specification languages that are structured to be familiar to those developing protocol standards, and tooling that can be used to generate parser code directly from standards documents. In this paper, we explore the effectiveness of these techniques for specifying real-world protocols within the Internet Engineering Task Force (IETF), one of the key standards development organisations for network protocols, by showing how they can be incorporated into the standard protocol specification for TCP [3].

Formal protocol description techniques and automated code generation do not, irrespective of whether they are easy to integrate with the standards development process, remove all of the main classes of parser bugs [4]. Parsing code is made insecure not just by ambiguous specifications, but also by the design of the protocol itself, and by the architecture of the code. Formal protocol description techniques can shape both of these. The expressivity of the protocol data description language determines the set of message formats that can be described, while the code generator determines the architecture, paradigm, and language of the parser code.

Accordingly, we consider the code generation functionality of our Network Packet Representation [2], and highlight those features of the representation that assist standards authors in writing clear, unambiguous specifications that generate well-formed code that is easy to reason about. We demonstrate the specification of the TCP packet format, show how its Network Packet Representation is derived and how code can then be generated from this representation. We show how the generated code can be integrated with an existing TCP implementation, and demonstrate its correctness and performance.

An increasing number of ad-hoc, semi-structured, protocol specification languages are seeing adoption within the standards process [5], [6]. This shows willingness within the standards community to experiment and improve their specifications. However, while adoption of these languages will lead to specifications that are more precisely written, precision on its own is not sufficient [4]. Effective protocol description languages must also limit expressiveness of the formats to those that can be safely parsed. In this paper, we demonstrate how the Network Packet Representation, in providing a common representation framework, can influence protocol design, and the architecture of generated implementations. Both of these have significant implications for the safety and trustworthiness of the documents and generated code.

We structure the remainder of this paper as follows. In Section II, we further describe the role of formal protocol description techniques and automatic code generators in determining the overall safety and trustworthiness of packet parsing code. Then, in Sections III, IV, and V, we step through the specification, representation, and code generation steps, respectively, for a description of TCP using the Network Packet Representation. In Section VI, we evaluate the generated code in terms of correctness and performance. Finally, Section VII describes the related work, and Section VIII concludes.

ISBN 978-3-903176-39-3 © 2021 IFIP

II. Motivation

The IETF standards development process has a goal of reaching “rough consensus” on the protocol specifications that it produces [7] [8]. This makes standardisation and protocol design a political, human-driven process, where ideas are exchanged and discussed, and draft documents are debated, both verbally, and in text-based forums such as mailing lists, GitHub issues, etc. As a result, documents are written primarily in English prose. While this allows the documents to be used in the consensus-building process, natural language can often be ambiguous and unclear, and this can lead to inconsistent and non-conforming implementations.

Given this problem, numerous formal description techniques have been developed to more precisely describe protocol behaviours and data formats. Systems such as TLA+ [9] and Alloy [10] can be used to describe and model protocol semantics, and have been used with some success in real-world systems [11], and many domain-specific formal protocol modelling languages also exist (e.g., [12], [13], [14]).

There is also a rich literature allowing for the format of the data that is exchanged by protocols to be expressed using formal, structured languages. Academic research in this space has focussed on protocol type systems, modelling protocol semantics in addition to data formats, in systems such as PADS [15], DataScript [16], PacketTypes [17], and the Meta Packet Language [18]. Specialised languages have also been developed, including YANG [19], a data modelling language, eTPL [20], an enhanced version of the TLS presentation language, and NetPDL [21], an XML-based packet description language.

As discussed by Reid et al. [22], overcoming usability concerns around formal methods is important for widespread adoption. To address these, the standards setting community has developed a number of domain-specific packet format description languages such as ABNF [23], ASN.1 [24], the TLS presentation language [5], the packet notation used in recent QUIC documents [6], and our Augmented Packet Header Diagrams [25]. These have generally been developed in an ad-hoc fashion, by contributors to the IETF that require a particular format to fit a specific use case. This means that, while they are only used in a limited set of documents, they lack the steep learning curve of the more formal protocol type systems. Their ad-hoc nature allows them to be developed and defined to meet the needs of different documents and groups. However, in being developed outside the context of formal languages and programming language design, these languages are often not sufficiently well-specified for machine-readability, crucial to enabling automatic code generation, and pay little attention to the expressiveness and power of the generated parsers.

Unlocking the benefits of machine-readability and code generation for these languages is best achieved by a common framework that enables the ease-of-use that ad-hoc formats bring, but that enforces the rigour that is required. In prior work, we developed the Network Packet Representation [2], a type system for protocol data, that aims to bridge the gap between the requirements of the standards community, for flexible languages that don’t alter the process, with the benefits that come from formal protocol descriptions. As shown in Figure 1, this has been developed to support multiple input protocol description languages, and to generate parser implementations in a range of output programming languages, to give flexibility of input syntax with a rigorous type system and code generation.

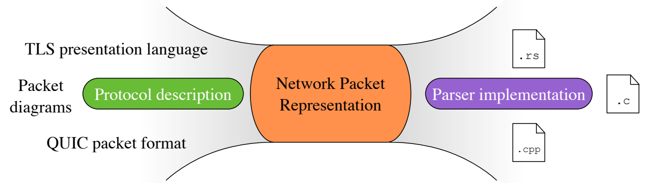


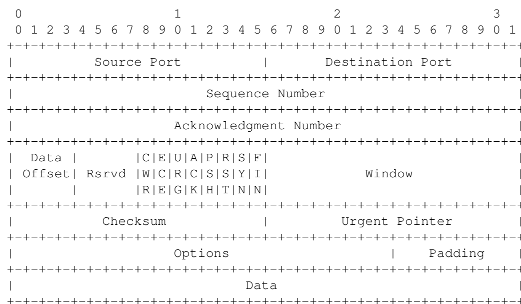
Figure 1: The Network Packet Representation allows parsers for protocol description languages to be decoupled from implementation generators.

In the following, we will show how the Network Packet Representation can be readily incorporated into the existing standards development workflow, to help in the specification of network protocols. We then consider how the representation system determines the set of protocols that it can represent. The complexity of the protocol’s syntax defines the inherent safety of the parsers that can be generated for it [26], with, for example, recursive definitions requiring parsers that are more powerful and less safe. We investigate how the protocol representation system can promote programming languages and paradigms that improve the security and reliability of the parsers that are generated. For example, the parser combinator paradigm [27] makes use of small, easy-to-debug parser functions. This paradigm, when combined with modern systems languages like Rust, can produce trustworthy and secure parsers.

Finally, we must also ensure that these features do not come at the expense of correctness or performance. We want to demonstrate that the Network Packet Representation can provide a significant shift in how protocols are developed and standardised, enabling automatic code generation directly from protocol standards documents, while producing code that is correct and performant. We will illustrate this by example, walking through the specification (§III), representation (§IV), and code generation (§V) phases for a description of TCP. We will then evaluate the generated parser in Section VI.

III. Specifying Protocol Data Formats

Automatically generating packet parser implementations from standards documents requires a description of the protocol syntax in a suitable protocol description language. As noted in Section II, there are both technical *and* social requirements on the design of such languages. Protocol description languages must be sufficiently well defined so that the documents that use them can be parsed, yet must be sufficiently flexible and expressive. Further, they must be human readable, usable by those with expertise in protocol design rather than formal methods, and support the consensus-based, discussion heavy, nature of the standardisation process [2]. The design and

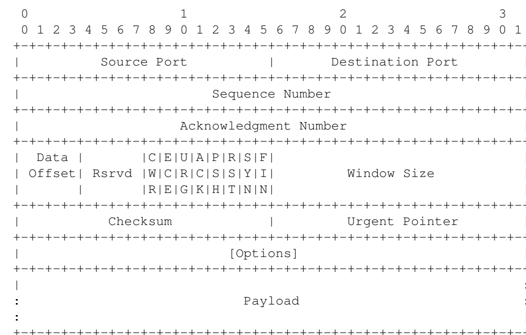


```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |       |C|E|U|A|P|R|S|F|                               |
| Offset| Rsrvd |W|C|R|C|S|S|Y|I|            Window             |
|       |       |R|E|G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             Data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
        Note that one tick mark represents one bit position.

                   Figure 1: TCP Header Format

Each of the TCP header fields is described as follows:

Source Port:  16 bits

  The source port number.

                       ...

Data Offset:  4 bits

  The number of 32 bit words in the TCP Header.  This indicates where
  the data begins.  The TCP header (even one including options) is an
  integral number of 32 bits long.

Rsrvd - Reserved:  4 bits

  A set of control bits reserved for future use.  Must be zero in
  generated segments and must be ignored in received segments, if
  corresponding future features are unimplemented by the sending or
  receiving host.

  The control bits are also know as "flags".  Assignment is managed
  by IANA from the "TCP Header Flags" registry [53].

Control Bits:  8 bits (from left to right) of currently assigned
control bits:

     CWR: Congestion Window Reduced (see [8])
     ECE: ECN-Echo (see [8])
     URG: Urgent Pointer field significant
     ACK: Acknowledgment field significant
     PSH: Push Function (see the Send Call description in
     Section 3.8.1)
     RST: Reset the connection
     SYN: Synchronize sequence numbers
     FIN: No more data from sender

                       ...

Options:  variable

                       ...
```

(a) draft-ietf-tcpm-rfc793bis-19 [3]

A TCP Segment is formatted as follows:



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |       |C|E|U|A|P|R|S|F|                               |
| Offset| Rsrvd |W|C|R|C|S|S|Y|I|          Window Size          |
|       |       |R|E|G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           [Options]                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               :
:                            Payload                            :
:                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

where:

Source Port: 16 bits. The source port number.

...

Data Offset (DOffset): 4 bits; **DOffset >= 5**. The number of 32 bit words in the TCP Header. This indicates where the data begins. The TCP header (even one including options) is an integral number of 32 bits long.

Reserved (Rsrvd): 4 bits; **Rsrvd == 0**. A set of control bits reserved for future use. Must be zero in generated segments and must be ignored in received segments, if corresponding future features are unimplemented by the sending or receiving host.

CWR: 1 bit. Congestion Window Reduced

ECE: 1 bit. ECN-Echo

URG: 1 bit. Urgent Pointer field significant

ACK: 1 bit. Acknowledgment field significant

PSH: 1 bit. Push Function (see the Send Call description)

RST: 1 bit. Reset the connection

SYN: 1 bit. Synchronize sequence numbers

FIN: 1 bit; **(FIN == 0) || (SYN == 0)**. No more data from sender.

...

Options: [TCP Option]; **Options#Size == (DOffset-5)*32**; present only when DOffset > 5. Options may occupy space at the end of the TCP header and are a multiple of 8 bits in length. All options are included in the checksum.

...

(b) Augmented Packet Header Diagram format [28]

Figure 2: Specifying the syntax of TCP segments ("..." indicates elided text; emphasis added to highlight constraints).

usability of protocol description languages, and associated tooling, is crucial to their adoption.

To illustrate this, we show how the TCP packet format can be specified using our Augmented Packet Header Diagrams protocol description language [25]. This is based on the informal diagrams that are already widely used in IETF documents, but updated with a limited set of changes to ensure consistency and support the machine-readability needed for automatic code generation. The familiarity of this description language is intended to encourage adoption within the standards development community.

We present an extract from the work-in-progress draft updating the TCP protocol specification in the IETF, dated 27 October 2020 (`draft-ietf-tcpm-rfc793bis-19`) [3], in Figure 2a. This is characteristic of many IETF standards, describing the packet format informally using a diagram of the packet headers followed by description of the fields in English prose. Figure 2b shows the same extract, translated into our Augmented Packet Header Diagram format, as has now been incorporated into the update to the TCP protocol specification, starting with the draft dated 3 May 2021 (`draft-ietf-tcpm-rfc793bis-21`). It is instructive to compare the two descriptions.

At a high-level, the version using the Augmented Packet Header Diagram is clearly, and intentionally, extremely close to that in the previous versions. The main elements remain the same: an ASCII diagram showing the layout of the packet is given, followed by a description list detailing each field. The diagram is useful for maintaining human-readability, while the description list allows the format to incorporate more machine-readable elements.

The need for machine-readability, to support automatic code generation and validation, requires that the Augmented Packet Header Diagram version be more precise, though. In this example, this is most apparent in the definition of the *Options* field. In Figure 2a, this is merely listed as being of variable

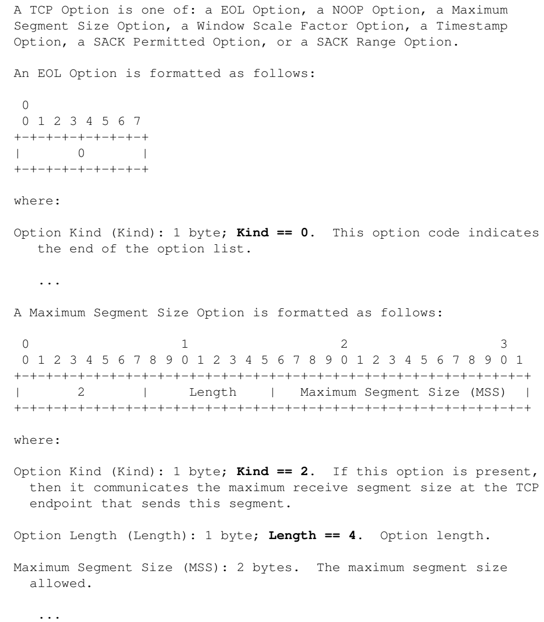
length. This version of the specification does not explicitly state the size of the field; it must be inferred from the definition of the *Data Offset* field. The Augmented Packet Header Diagram definition of the same field must be more explicit. As shown in Figure 2b, the *Options* field has a length specified as an expression (`Options#Size == (DOffset-5)*32`) and a constraint that it is only present when `DOffset > 5`. The Augmented Packet Header Diagram has a rich expression grammar that is used to express constraints on the values and lengths of fields; these are emphasised in bold in the extracts shown in Figure 2b and Figure 3. While these constraints are needed to form a precise definition of the types that are described, they also greatly improve human readability.

More generally, the Augmented Packet Header Diagram language sits on top of the type system of the Network Packet Representation. As will be explored in more depth in Section IV, field are strongly typed. For most of the fields in the example, these are `BitString` types, defined in terms of their name and width. However, the *Options* field is an `Array` type, as indicated by the square brackets given in both the diagram and description list, holding variants of an `Enum` to describe the different options. The definition of this field in Figure 2b indicates that the field is an array of *TCP Option* instances, where the *TCP Option* type is described later in the document using the form shown in Figure 3 to describe the variants.

Importantly, the Augmented Packet Header Diagram format is not just a formalism of the existing ad-hoc syntax, but *also* constrains the set of protocols that can be defined, affecting the safety and decidability properties of the parsers that can be generated. An important boundary in terms of decidability is the inclusion of recursively-defined elements. These elements lead to protocol syntax that is undecidable [26], and for which safe parsers are difficult or impossible to write. The Network Packet Representation cannot be used to express recursively-defined types, and so all languages that are used to generate it must similarly limit the protocols that can be expressed. This demonstrates the power of having a common protocol representation system: it influences the design and safety of protocol description languages, and so of protocols themselves.

In summary, the Augmented Packet Header Diagram format is intentionally close to the ad-hoc formats used in standards documents today, while being more precise and machine-readable. This familiarity eliminates the steep learning curves associated with other description techniques, removing one of the main barriers to adoption. The use of this format in recent drafts of the TCP specification produced by the IETF [3] validates our approach – and shows the usability of our design. The format is similar enough to existing practice that it can be understood and adopted by those developing protocol standards, without the need for extensive training and education.

We have further written draft specifications for the format of UDP datagrams (see [29]) and QUIC packets and frames (see [30]) using Augmented Packet Header Diagrams. These are omitted here due to space constraints, but show that the format can be used to express the syntax of a wide range of protocols, including both traditional and widely deployed protocols such as TCP and UDP, and also modern designs with integrated security and more complex packet formats, such as QUIC.



```
A TCP Option is one of: a EOL Option, a NOOP Option, a Maximum
Segment Size Option, a Window Scale Factor Option, a Timestamp
Option, a SACK Permitted Option, or a SACK Range Option.

An EOL Option is formatted as follows:

 0
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|       0       |
+-+-+-+-+-+-+-+-+

where:

Option Kind (Kind): 1 byte; Kind == 0.  This option code indicates
   the end of the option list.

   ...

A Maximum Segment Size Option is formatted as follows:

 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       2       |     Length    |   Maximum Segment Size (MSS)  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

where:

Option Kind (Kind): 1 byte; Kind == 2.  If this option is present,
  then it communicates the maximum receive segment size at the TCP
  endpoint that sends this segment.

Option Length (Length): 1 byte; Length == 4.  Option length.

Maximum Segment Size (MSS): 2 bytes.  The maximum segment size
  allowed.

   ...
```

Figure 3: Specifying TCP options using the Augmented Packet Header Diagram format [28] ("..." indicates elided text; emphasis added to highlight constraints).

IV. Typed Protocol Representation

The Network Packet Representation [2] is a type system and common intermediate representation for describing network protocol data. It provides a fixed set of internal types, along with type constructors for various representable types used in concrete protocols. The internal types comprise abstract `Boolean` and `Number` types, functions, a type to represent a `Protocol`, and a parsing `Context`. The representable types comprise `BitString` types that represent single protocol fields of a known size; `Array` types that represent sequences of such fields; a parameterised `Option<T>` type that represents an optional field of type `T` with an associated boolean expression that indicates presence or absence of the field; product types, `Struct`, that represent PDUs, or fragments of a PDU, comprised of a combination of fields; and `Enum` types that represent alternatives. Fields in `Struct` types are associated with optional expressions that constrain their size or value dependent on other fields. The type system is described in more detail in [2]. The Network Packet Representation encodes the protocol data units, and contextual information needed to parse those PDUs, in a strongly typed hierarchical data structure. This is organised in a top-down manner, starting with the `Protocol` type, parameterised by the `Context` and an `Enum` representing the possible PDUs, then defining the `Struct` types that represent each PDU in turn.

Figure 4 shows the types constructed in the Network Packet Representation of TCP, derived from parsing a specification written using the Augmented Packet Header Diagram format

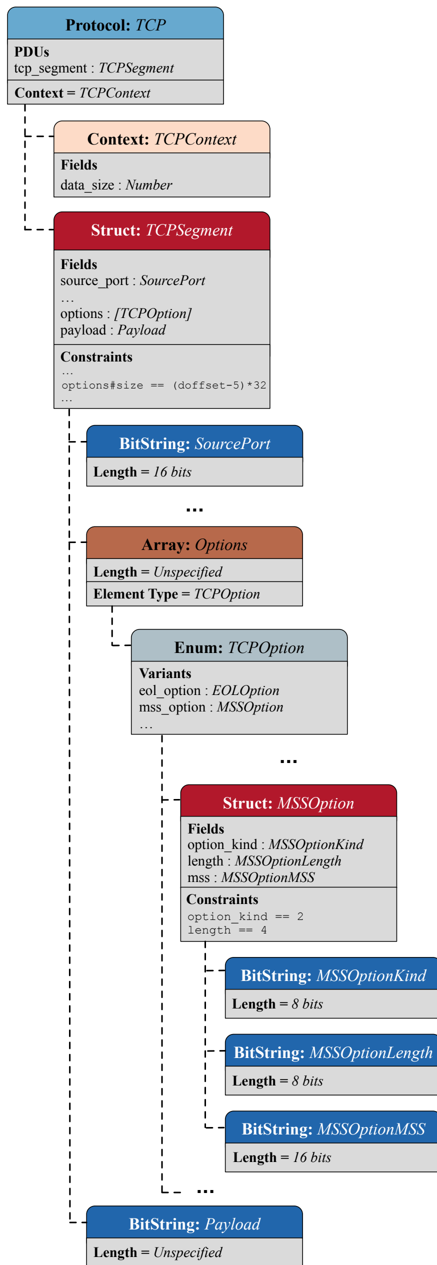


Figure 4: The Network Packet Representation for TCP, as described in Figure 2b ("..." indicates components omitted for brevity).

discussed in Section III. At the top-level, this comprises a *TCP* type that is an instance of a `Protocol`, with type parameters indicating that the `Context` is of type *TCPContext* and that there is a single PDU type, *TCPSegment*.

The *TCPContext* is a product type that represents persistent state that can be accessed throughout the parsing process. TCP requires no special parsing context, so the only field on this type is *data_size*, the size of the datagram being parsed. The context will be discussed further in Section V.

The *TCPSegment* is a `Struct` type the represents the format of a TCP segment, the sole PDU in the TCP protocol. A TCP segment comprises a sequence of fields, starting with the source and destination ports, sequence and acknowledgement numbers, and so on. For each of these, a field is added to the *TCPSegment* type, and a new type is generated to represent that field. This begins with the Source Port field, for which a new `BitString` type, *SourcePort*, is created with a length of 16 bits. Then, a field *source_port*, with type matching the newly created *SourcePort* type, is added to the *TCPSegment* definition. This process is repeated for the remaining fields, defining appropriate types to represent each.

Most fields in a TCP segment are straightforward, with fixed size. However, as seen in Figure 2b, the *Options* field is an `Array` containing elements of the type *TCPOption*. As shown in Figure 3, a TCP option can take on one of multiple different formats. As a result, *TCPOption* is constructed as an `Enum` type with several different variants: *EOLOption*, *MSSOption*, and so on. Each of the variants is a `Struct` type, itself holding fields defined to be appropriate types, with constraints on field values. For example, the *MSSOption* type is dependently typed with constraints that its *option_kind* has value 2 and its *length* is four bytes. Constraints are declarative and guide parsing, as we discuss in Section V.

The *TCPSegment* itself also has constraints. For example, the total size of the *Options* array is *(doffset-5)*32* bytes, as shown in Figure 2b and reflected in the type definition in Figure 4. The Network Packet Representation has a rich expression grammar that enables the value of fields to be used in place of fixed values. The expression grammar allows for the values of previously defined fields to be accessed, and combined with internal types, like numbers, using typical arithmetic operators. These expressions form part of the definition of a type: binary data is only a *TCPSegment* if, when parsed, it contains all of the fields of the type *and* if all of the constraints are true.

The final field of the TCP segment is the *Payload* field. This is a `BitString` type, but it does not have a fixed size, or a known size that can be defined by an expression. As a result, the *Payload* type is constructed with unspecified size, leaving it to be inferred by the Network Packet Representation tooling.

The size inference process generates an expression for the size of types with unspecified sizes. This expression is parameterised by a value in the *TCPContext*, *data_size*, that is set to the length of the segment being parsed by the generated parsing code. In this example, *data_size* would be the entire length of the TCP segment. The size inference process is recursive, beginning with each type specified as a PDU in the *Protocol* definition. The size inference algorithm apportions *data_size* to types with unspecified lengths intuitively, recursively replacing unspecified lengths with expressions as it traverses the set of types that define a protocol. For example, the *Payload* type would have a length that is specified as *data_size*, less the sum of all of the other fields. An important constraint results from this algorithm: `Struct` types can contain at most one field type whose size is unspecified. If multiple fields had unspecified sizes, then it would not be possible to determine the size of each field.

This section has highlighted the importance of an abstract protocol type system, like that of the Network Packet Representation, in removing a number of the properties of some protocol designs that are inherently unsafe. In addition to preventing recursive definitions, as described in Section III, the Network Packet Representation requires that types and expressions only be parameterised in terms of previously defined types, and ensures that all representable types have known sizes. In Section V, we will show how these properties of the Network Packet Representation influence the quality, security, and trustworthiness of the parser implementations that are generated from it.

V. Code Generation

The final component of the architecture of the Network Packet Representation is code generation. It has been shown that the programming language and paradigm used by parser code has important consequences for security [31] [32]. The structure of the Network Packet Representation, in determining the design of code generators, can promote the adoption of safer languages and paradigms, improving the security of generated implementations. As illustrated in Figure 4, the Network Packet Representation is constructed from a top-down traversal of a protocol's definition, with independent types being constructed before the complex types that depend on them, resulting in a tree structure. This structure mirrors the parser combinator idiom [27], where small unit parsers are combined into parsers for more complex types. Just as types in the Network Packet Representation may only depend on previously defined types, parser combinator functions may only make use of previously specified functions. This means that a bottom-up mirror of the traversal used to construct the Network Packet Representation is needed to generate parser combinator code. This traverses the type graph for the `Protocol`, traversing those types listed as PDUs, and generates parser code for the leaf types first, then code that parses the containing types, and so on until the entire protocol can be parsed.

We have developed a prototype implementation that generates protocol parsers, written in Rust, using the `nom` library [33], from the Network Packet Representation, using Augmented Packet Header Diagrams as an input format.[1] In this section, we will describe the output for our TCP example, and give snippets of the generated code to illustrate the benefits of the parser combinator approach, and the ways in which the architecture of the generated code improves safety and reliability.

Our code generator creates a TCP parser implementation that is comprised of around 1115 lines of Rust code. This is generated from the Augmented Packet Header Diagram description of TCP [28], as given in Figures 2b and 3, and has support for several TCP options, including MSS, timestamp, and SACK options. At the top-level, the generated code provides a `parse_pdu` function that takes an array of bytes as input, and, if successful, returns a *TcpSegment*.

[1]A snapshot of our parser generator, matching this paper, is available at http://dx.doi.org/10.5525/gla.researchdata.1139 while the main repository is at https://github.com/glasgow-ipl/ips-protodesc-code

```
#[derive(Debug, PartialEq, Eq)]
pub struct TcpSegmentSourcePort(pub u16);

#[inline]
pub fn parse_tcp_segment_source_port<'a>(input: (&'a [u8], usize),
                                          context: &'a mut Context)
        -> (IResult<(&'a [u8], usize), TcpSegmentSourcePort>, &'a mut Context) {
    (take(16 as usize)(input).map(|(i, o)| (i, TcpSegmentSourcePort(o))), context)
}
```

Figure 5: Generated code for the *TcpSegmentSourcePort* type.

```
#[derive(Clone, Debug, PartialEq, Eq)]
pub struct TcpSegment {
    pub source_port: TcpSegmentSourcePort,
    pub destination_port: TcpSegmentDestinationPort,
    pub sequence_number: TcpSegmentSequenceNumber,
    pub acknowledgment_number: TcpSegmentAcknowledgmentNumber,
    pub data_offset: TcpSegmentDataOffset,
    pub reserved: TcpSegmentReserved,
    pub cwr: TcpSegmentCwr,
    pub ece: TcpSegmentEce,
    pub urg: TcpSegmentUrg,
    pub ack: TcpSegmentAck,
    pub psh: TcpSegmentPsh,
    pub rst: TcpSegmentRst,
    pub syn: TcpSegmentSyn,
    pub fin: TcpSegmentFin,
    pub window_size: TcpSegmentWindowSize,
    pub checksum: TcpSegmentChecksum,
    pub urgent_pointer: TcpSegmentUrgentPointer,
    pub options: Option<TcpSegmentOptions>,
    pub payload: TcpSegmentPayload,
}

#[inline]
pub fn parse_tcp_segment<'a>(mut input: (&'a [u8], usize),
                             mut context: &'a mut Context)
            -> (IResult<(&'a [u8], usize), TcpSegment>, &'a mut Context) {
    let source_port = match parse_tcp_segment_source_port(input, context) {
        (IResult::Ok((i, o)), c) => {
            input = i;
            context = c;
            o
        }
        (IResult::Err(e), c) => return (IResult::Err(e), c),
    };

    ...

    let data_offset = match parse_tcp_segment_data_offset(input, context) {
        (IResult::Ok((i, o)), c) => {
            // check constraint: (data_offset >= 5)
            if !((o.0 >= 5)) {
                return (IResult::Err(Err::Error((input, ErrorKind::NonEmpty))), c);
            };
            input = i;
            context = c;
            o
        },
        (IResult::Err(e), c) => return (IResult::Err(e), c),
    };

    ...

    (IResult::Ok((input, TcpSegment { ... }, )), context)
}
```

Figure 6: Generated code for the *TcpSegment* type ("..." indicated elided text).

Below this top-level API, the generated code is structured around small parser combinator functions. Figure 5 illustrates this, showing the type and parser function that is generated for the *TcpSegmentSourcePort* type. For brevity, a full discussion of the details of the Rust syntax is omitted. However, the generated parser combinator takes as input a tuple that has a byte array and offset (i.e., how many bits of the array have been consumed), and a context type. As noted in Section IV, this context can be accessed throughout the parsing process. The function returns a result type, containing a tuple with an input stream and offset – possibly with bits consumed – and, if successful, the constructed type. The context type is also returned.

Similar parser combinator functions are generated for each of the types given in the Network Packet Representation. An important part of the parser combinator paradigm is that simple parser functions are combined together into parsers

for more complex types. This means that most functions are short, making them easier to understand and debug [32]. The simplest function is the `take` function, as used in Figure 5. This is a `nom` function that removes the specified number of bits from the input stream, and returns the stream with the bits consumed. To illustrate how the generated parser functions are combined, Figure 6 shows the Rust type and code generated for the *TcpSegment* type. In this example, the previously generated types and functions are composed together. This composition differs based on the type. For `Struct` types, parsers are composed sequentially, based on the type of each field; for `Enum` types, parsers are tried in turn until one is successful; and for `Array` types, a parser is called repeatedly as needed. In all cases, parsing may fail, and an error is propagated upwards from the base parser.

An important part of this is the use of expressions and constraints. In Figure 6, the constraint on the value of the parsed *data_offset* field is checked. This constraint was given using the Augmented Packet Header Diagram description in Figure 2b; it was then used to define the Network Packet Representation for TCP, shown in Figure 4; and finally, the constraint is expressed in the code, where parsing will fail if the constraint is not met. This example, while relatively straightforward, demonstrates the power of the overall approach.

Momot et al. [4] describe a taxonomy of common parser bugs. Two categories in this taxonomy – *shotgun parsing* (or ad-hoc validation in the parsing process) and *non-minimalist input-handling code* – are related to the tendency to produce parser code that is overly complex, and that combines the simpler task of parsing the protocol's syntax with the more difficult step of semantic validation. Separating these concerns allows for code that is simpler and easier to debug. The generated code described in this section is narrowly designed to provide a complete parser for the specified protocol. The Network Packet Representation does not capture complex semantic logic for the protocol, and includes only the information needed to parse and validate incoming TCP segments. This scope means that the generated code is simple and easy to debug.

The Network Packet Representation can be used to model the stateless semantics of a protocol. For example, as shown in Figure 2b, constraints can be used to ensure that incoming segments do not have both the SYN and FIN flags set. However, the parser cannot check if an incoming segment's flags are set appropriately within a particular flow. For example, it is not valid for a SYN packet to arrive after a RST packet. Checking this would require the parser to maintain a state machine for the protocol, which is not captured by the Network Packet Representation. This means that our parser is not standalone: it is designed for inclusion within a larger implementation that includes validation of the protocol's semantic logic. It is within this context that we evaluate the correctness and performance of our generated TCP parser code in Section VI.

VI. Evaluation

There are two broad requirements for the code generated from the Network Packet Representation: it must be correct, in so far as it matches the specification, and it must be sufficiently performant, so as to enable interoperability testing and test-driven development. While a level of usable performance is required, the generated code is designed to act as a sample implementation. Other non-functional properties of the code, such as readability, take precedence.

In this paper, we have described how a specification of TCP written in the Augmented Packet Header Diagram language [28] (Figures 2b and 3) can be used to derive a Network Packet Representation (Figure 4), and from this, a Rust parser implementation (Figures 5 and 6). As the generated code provides only a parser implementation, we must integrate it with a larger TCP implementation to evaluate its correctness and performance.

To do this, we have integrated the generated code with `smoltcp`[2], replacing its parser with our generated implementation. `smoltcp` is a standalone TCP/IP stack written in Rust, and is designed to be performant for embedded, real-time systems. The implementation includes support for many of TCP's features, including checksum validation, window scaling, and maximum segment size negotiation.

`smoltcp`'s extensive documentation and layered architecture made it straightforward to integrate our generated parser with the wider implementation. It is instructive to consider the implementation that our generated code replaces. The existing implementation makes use of bitmasks to parse the TCP header. This is a common, but brittle and error-prone, approach.

Integrating with `smoltcp` provides us with a suite of example applications, including a minimal clone of *tcpdump* and a benchmarking script. We use these tools to evaluate the correctness and performance of the generated code.

A. Correctness

To evaluate the correctness of the generated code, we construct a set of TCP segments based upon the standards document that defines the protocol. These segments have been constructed using *scapy* and captured as `pcap` files. To ensure that the test `pcap`s have been generated correctly, they have been inspected using *tcpdump*. The correctness of the generated code is evaluated by running `smoltcp`'s minimal *tcpdump* clone over the test files, and comparing the output with that of *tcpdump* itself.

Table 7 lists the test segments that were constructed. In total, 25 tests were performed. These included setting each field to a known value, and ensuring that this value was parsed correctly. All of the supported TCP options, including the Timestamp and SACK block options, were tested. Finally, the set of test segments included five that were deliberately invalid, to test both the syntactic checks of the generated parser, and the semantic checks of `smoltcp`.

All of the test segments were handled by the implementation as expected. The generated parser code extracted the values of each field correctly, and the integration with `smoltcp` used these values properly. This extended to the tests that contained

[2]https://m-labs.hk/software/smoltcp/

#	Fields set	Parsed?
1	$SourcePort = 8080$	Y
2	$DestinationPort = 9999$	Y
3	$SeqNum = 1200$	Y
4	$AckNum = 5000$	Y
5	$CWR = 1$	Y
6	$ECE = 1$	Y
7	$URG = 1$	Y
8	$PSH = 1$	Y
9	$RST = 1$	Y
10	$SYN = 1$	Y
11	$FIN = 1$	Y
12	$Window = 2000$	Y
13	$UrgentPointer = 10$	Y
14	$Ops = [EOLOp]$	Y
15	$Ops = [NoOpOp, EOLOp]$	Y
16	$Ops = [MSSOp(1200)]$	Y
17	$Ops = [TSOp(val = 20, ecr = 10)]$	Y
18	$Ops = [SAckOKOp]$	Y
19	$Ops = [SAck(l = 1000, r = 2000)]$	Y
20	$Payload = "Hello, world"$	Y
21	$Checksum = 15$	Y smoltcp indicated invalid checksum.
22	$Ops = [Option(kind = 34)]$	N Invalid option kind.
23	$DataOffset = 1$	N $DataOffset < 5$.
24	$SYN = 1; FIN = 1$	N Invalid flags.
25	$Reserved = 12$	N $Reserved \neq 0$.

Figure 7: Correctness tests; fields set to known values as indicated, with all other fields set to acceptable values. $SYN = 1$ in all tests, except where other flags set explicitly.

invalid values. In test #21, the checksum was set to a known value. While this segment was parsed, it failed `smoltcp`'s semantic validation checks. This demonstrates the separation of parsing and validation that was described in Section V. In test #22, an invalid option was added to the TCP segment. This option was not included in the description of the protocol (Figure 3), and so it could not be parsed. The remaining tests (i.e., # 23 through 25) check that the other constraints given in the protocol's definition (Figure 2b) are enforced.

B. Performance

While the correctness of the generated parser code is paramount, it is also the case that it must be sufficiently performant. The performance of parser implementations is often a trade-off with other attributes: tightly written code is often less readable. The generated code must be readable to act as a sample implementation. However, if the code is too slow, then it is unlikely to be used, undermining its purpose.

To evaluate the performance of our generated implementation, we compare it with the existing `smoltcp` parser that it replaces. This means that all other factors remain the same. The evaluations consisted of running `smoltcp`'s benchmarking script. The test was run on a machine with an Intel Xeon E3 1240 CPU with 32GB of RAM. Both codebases were compiled using rustc 1.43.0. Finally, as shown in Figure 6, the generated code includes hints to the Rust compiler that it should inline the parsing functions.

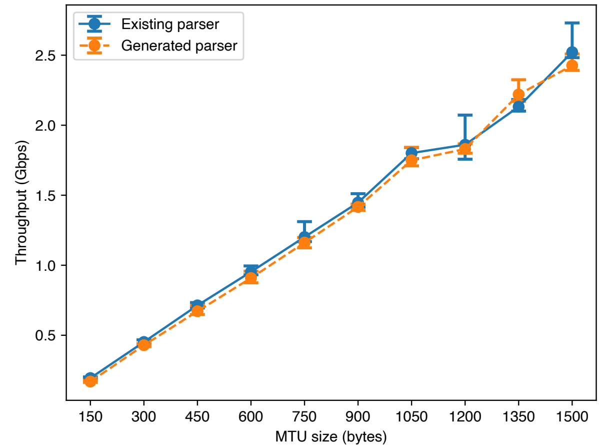


Figure 8: Throughput of `smoltcp`'s existing parser, compared with the generated parser, for various MTU sizes.

The benchmarking script used the loopback interface to transmit 1000MB of data, and to parse the incoming data. The script was run using different MTU values (from 150 bytes to 1500 bytes, in 150 byte increments). Each run was repeated 10 times for each implementation.

Figure 8 shows the results of the performance evaluation. With an MTU of 1500 bytes, the average throughput of the existing implementation was 2.52Gbps, compared to 2.43Gbps for the generated implementation, a 3.77% reduction. When the MTU was reduced to 150 bytes, average throughput of the existing implementation dropped to 0.2Gbps, and to 0.17Gbps for the generated parser. This demonstrates that, even when the header comprises a larger portion of the packet, the generated parser still performs relatively well.

These tests have shown that our generated TCP parser implementation is correct and performant, capable of processing incoming TCP segments in the order of gigabits per second as typical MTU sizes. These are important results. They show that the benefits of being able to derive an implementation directly from standards documents can be unlocked while maintaining the correctness and performance of the code.

VII. Related Work

The Network Packet Representation [2] builds upon more recent representation systems that can capture the external syntax of protocols, and a model of their internal representation, essential for representing modern, complex protocols. These include Nail [34], which builds upon Hammer [35], and produces parser combinators for grammars specified in domain-specific languages, using stream transformations and arenas to support complex protocols. Narcissus [36] supports multi-stage parsing with a framework for the Coq proof assistant.

The goals of adopting a representation system that integrates with the standardisation process is rooted both within the LangSec community, which calls for protocols to be treated as formal input languages [37], and the need to adapt formal methods to users [22]. Momot et al. [4] describe common parser vulnerabilities, and suggest remedies for them. As we have shown, the architecture of the Network Packet Representation promotes and enforces a number of these remedies. The type system ensures that protocol specifications are complete, that the complexity of protocol syntax is minimised, and that the generated code separates the functions of input parsing with semantic validation.

VIII. Conclusions

We have demonstrated the benefits of the approach advocated by the Network Packet Representation, shifting towards the use of tooling that can automatically generate a sample protocol parser implementation from a standards document that specifies it, can be achieved while maintaining correctness and performance. We have shown that a description of TCP, in a format broadly similar to that in typical use, can be used to generate a safe, correct parser implementation that is capable of processing incoming data at a rate of gigabits per second.

We believe that this architecture provides a solid foundation for future work. With a formal representation of the protocol's syntax, it should be possible to generate test cases for that protocol. This would add to the tooling demonstrated in this paper, and contribute to a step change in how protocols are developed, helping to further improve the security and trustworthiness of Internet protocol standards.

Acknowledgements

This work is funded by the UK Engineering and Physical Sciences Research Council, under grant EP/R04144X/1.

References

[1] S. Bratus, M. L. Patterson, and A. Shubina, "The bugs we have to kill," *;login:*, vol. 40, no. 4, pp. 4–10, Aug. 2015.
[2] S. McQuistin, V. Band, D. Jacob, and C. Perkins, "Parsing protocol standards to parse standard protocols," in *Proceedings of the Applied Networking Research Workshop*, 2020, pp. 25–31.
[3] W. Eddy, "Transmission Control Protocol (TCP) Specification," Internet Engineering Task Force, May 2021, Work in progress.
[4] F. D. Momot, S. Bratus, S. M. Hallberg, and M. L. Patterson, "The seven turrets of Babel: A taxonomy of LangSec errors and how to expunge them," in *Cybersecurity Development (SecDev)*. Boston, MA, USA: IEEE, Nov. 2016. DOI:10.1109/SecDev.2016.019
[5] E. Rescorla, "The transport layer security (TLS) protocol version 1.3," Internet Engineering Task Force, Mar. 2018, RFC 8446.
[6] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," Internet Engineering Task Force, Jan. 2021, Work in progress.
[7] D. D. Clark, "A cloudy crystal ball - visions of the future," in *Proceedings of the Internet Engineering Task Force*, vol. 24, Cambridge, MA, USA, Jul. 1992, pp. 539–543. [Online]. Available: http://www.ietf.org/proceedings/24.pdf
[8] P. Resnick, "On consensus and humming in the IETF," Internet Engineering Task Force, Jun. 2014, RFC 7282.
[9] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Jul. 2002.
[10] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Nov. 2011.
[11] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How Amazon Web Services uses formal methods," *Communications of the ACM*, vol. 58, no. 4, pp. 66–73, Apr. 2014.
[12] T. G. Griffin and J. L. Sobrinho, "Metarouting," in *Proceedings of the SIGCOMM Conference*. Philadelphia, PA, USA: ACM, Aug. 2005.
[13] A. Wang, L. Jia, C. Liu, B. T. Loo, O. Sokolsky, and P. Basu, "Formally verifiable networking," in *Proceedings of the Workshop on Hot Topics in Networks*. New York, NY, USA: ACM, Oct. 2009.
[14] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "p4v: practical verification for programmable data planes," in *Proceedings of the SIGCOMM Conference*. Budapest, Hungary: ACM, Aug. 2018, pp. 490–503.
[15] K. Fisher and R. Gruber, "PADS: a domain-specific language for processing ad hoc data," in *Proc. PLDI*. Chicago, USA: ACM, 2005.
[16] G. Back, "Datascript-a specification and scripting language for binary data," in *International Conference on Generative Programming and Component Engineering*. Pittsburgh, PA, USA: Springer, 2002.
[17] P. J. McCann and S. Chandra, "Packet types: abstract specification of network protocol messages," *ACM SIGCOMM Computer Communication Review*, vol. 30, no. 4, pp. 321–333, 2000.
[18] A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan, "Melange: Creating a "functional" internet," in *Proc. EuroSys*. Lisbon, Portugal: ACM, 2007, pp. 101–114.
[19] M. Bjorklund, "YANG – a data modeling language for the network configuration protocol (NETCONF)," Internet Engineering Task Force, Oct. 2010, RFC 6020.
[20] A. Walz and A. Sikora, "eTPL: An enhanced version of the TLS presentation language suitable for automated parser generation," in *International Conference on Intelligent Data Acquisition and Advanced Computing Systems*, vol. 2. IEEE, 2017, pp. 810–814.
[21] F. Risso and M. Baldi, "NetPDL: an extensible XML-based language for packet header description," *Computer Networks*, vol. 50, no. 5, pp. 688–706, 2006.
[22] A. Reid, L. Church, S. Flur, S. De Haas, M. Johnson, and B. Laurie, "Towards making formal methods normal: meeting developers where they are," in *Proceedings of the Workshop on Human Aspects of Types and Reasoning Assistants*, Online, Oct. 2020.
[23] D. Crocker, "Augmented BNF for syntax specifications: ABNF," Internet Engineering Task Force, Jan. 2008, RFC 5234.
[24] I. T. Union, "Abstract syntax notation one (ASN.1): Specification of basic notation," ITU-T Recommendation X.680, Aug. 2015.
[25] S. McQuistin, V. Band, D. Jacob, and C. S. Perkins, "Describing Protocol Data Units with Augmented Packet Header Diagrams," Internet Engineering Task Force, May 2021, Work in progress.
[26] L. Sassaman, M. L. Patterson, S. Bratus, and A. Shubina, "The halting problems of network stack insecurity," *USENIX; login*, vol. 36, no. 6, pp. 22–32, 2011.
[27] W. H. Burge, "Recursive programming techniques," 1975.
[28] S. McQuistin, V. Band, D. Jacob, and C. S. Perkins, "Describing TCP with Augmented Packet Header Diagrams," Internet Engineering Task Force, May 2021, Work in progress.
[29] ——, "Describing UDP with Augmented Packet Header Diagrams," Internet Engineering Task Force, May 2021, Work in progress.
[30] ——, "Describing QUIC's Protocol Data Units with Augmented Packet Header Diagrams," Internet Engineering Task Force, May 2021, Work in progress.
[31] E. Jaeger, O. Levillain, and P. Chifflier, "Mind your language (s)," in *1st LangSec workshop of IEEE Security & Privacy*, 2014.
[32] P. Chifflier and G. Couprie, "Writing parsers like it is 2017," in *2017 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2017, pp. 80–92.
[33] G. Couprie, "Nom, a byte oriented, streaming, zero copy, parser combinators library in rust," 05 2015, pp. 142–148.
[34] J. Bangert and N. Zeldovich, "Nail: A practical tool for parsing and generating data formats," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, 2014, pp. 615–628.
[35] M. Patterson and D. Hirsch, "Hammer parser generator," 2014. [Online]. Available: https://github.com/UpstandingHackers/hammer
[36] B. Delaware, S. Suriyakarn, C. Pit-Claudel, Q. Ye, and A. Chlipala, "Narcissus: correct-by-construction derivation of decoders and encoders from binary formats," *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, pp. 1–29, 2019.
[37] L. Sassaman, M. L. Patterson, S. Bratus, and M. E. Locasto, "Security applications of formal language theory," *IEEE Systems Journal*, vol. 7, no. 3, pp. 489–500, 2013.