Parsing Protocol Standards to Parse Standard Protocols

Stephen McQuistin
University of Glasgow
sm@smcquistin.uk

Vivian Band
University of Glasgow
vivianband0@gmail.com

Dejice Jacob
University of Glasgow
d.jacob.1@research.gla.ac.uk

Colin Perkins
University of Glasgow
csp@csperkins.org

ABSTRACT

Internet protocol standards have been slow to adopt formal protocol description languages and methodologies, and are still largely written as English prose. This makes it hard to check them for correctness, or to automatically derive implementations from standards. Reasons for this are both technical and social. Some methodologies effectively describe complex communication patterns, but cannot model protocol data. Others are unnecessarily tied to particular description formats, or use unfamiliar concepts and terminology, and don't address usability by standards developers.

We assess the viability of existing approaches to modelling and parsing protocol data, and identify missing features needed to represent emerging protocols. We present a typed protocol representation that can describe: (i) the format of protocol data, including data-dependent formats; (ii) contextual information needed to maintain parser state, where correct parsing may depend on out-of-band information or prior packets; and (iii) transformations and helper functions needed for multi-stage parsing. We discuss social barriers to adoption, and describe a set of principles to encourage use of formal languages within the Internet standards process. We show how to integrate our approach with the existing standards process, using QUIC as an example.

CCS CONCEPTS

• **Networks → Protocol correctness**; • **Software and its engineering → Domain specific languages**.

ACM Reference Format:
Stephen McQuistin, Vivian Band, Dejice Jacob, and Colin Perkins. 2020. Parsing Protocol Standards to Parse Standard Protocols. In *Applied Networking Research Workshop (ANRW '20), July 27–30, 2020, Online (Meetecho), Spain.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3404868.3406671

ANRW '20, July 27–30, 2020, Online (Meetecho), Spain

1 INTRODUCTION

The process by which Internet protocols are standardised is largely centred around documents written in English prose. To some extent, this is desirable: prose documents are useful for exchanging ideas, facilitating discussion, and building consensus. However, as protocols become more complex, the limitations of this approach become clear. Inconsistencies and ambiguities are easily introduced into the standards, making it difficult to develop implementations that conform to the specification. Use of formal specification languages would make the standards documents more machine readable. This would make them easier to test, and would help to support, for example, automatic generation of packet parser code from the specification. Use of such formalisms is, however, not common in Internet protocol standards.

There are technical and non-technical reasons for the slow adoption of formal description techniques by the Internet standards community. Technical limitations include protocol description languages that cannot fully describe the syntax of modern protocols. Weaknesses of current formal descriptions include formalisms that effectively model abstract communication patterns, but cannot describe the protocol data being exchanged. On the non-technical side, models may tightly integrate with unfamiliar protocol description languages or assume familiarity with concepts that are not widely known outside the formal modelling community. Moreover, adoption of new techniques requires engineers developing protocol standards to learn new skills for seemingly uncertain future benefits, and to overcome organisational inertia.

If the Internet standards development community is to adopt formal protocol description and modelling techniques, to help ensure correctness of its protocol specifications, then those techniques will need to be usable within the existing standards development process, and will need to be usable by existing standards developers. In this paper, we consider one part of this problem: how to describe protocol data, and how to parse and serialise that data into packets. Importantly, we show how to do this in a way that integrates within the IETF standards process, in a general-purpose manner that allows for a range of protocol description formats to be used.

We describe current approaches to formal description of protocols and their weaknesses. We then describe an expressive and extendable type system that enables most binary protocols to be expressed, with support for strongly typed transformation functions and parsing contexts. We also address the social barriers to the adoption of formal techniques, with a flexible, language-agnostic framework, and principles for using these techniques in standards documents. Finally, we outline a simple packet header diagram language, and show how this can used to express the QUIC protocol [11].

2 DESCRIBING PROTOCOL DATA

Protocols exchange data. It is crucial that protocol data units (PDUs) exchanged by a protocol are well described. This includes both their syntax, to allow for automated generation of parsing and serialisation code, and their semantics, in the form of a strongly typed representation that allows for modelling and validation of the protocol.

Syntax description languages include ABNF [7], ASN.1 [16], and the TLS 1.3 presentation language [17]. These are metalanguages, and are used to formally describe the syntax of protocols. Each language lends itself to the description of different protocols: ABNF, for example, is widely used to define textual protocols, while ASN.1 provides support for various encodings and is used to define binary formats. That different languages serve different purposes and communities is not generally problematic. Yet, these languages can only be used to describe the syntax of protocols. This is beneficial, but incomplete: it is necessary to capture the semantics of a protocol to allow for modelling and validation.

Several **protocol type systems** have also been developed to model the semantics of the data being exchanged, and their use shows the benefits of formal protocol description. eTPL [24], an enhanced version of the TLS presentation language, has been used to generate code that enables the automatic detection of security vulnerabilities in TLS implementations [25]. YANG [4], a data modelling language, allows for the complete description of all data exchanged, and enables conformance testing and validation. NetPDL [18], an XML-based packet header description language, has been used to generate performant packet processing code [2].

More generalised approaches have also been developed, such as PADS [10], DataScript [1], PacketTypes [13], and the Meta Packet Language (MPL) [12]. These allow much of the semantics of the protocol data to be captured, using a basic set of types, including bit strings, arrays, and structure types.

Protocols, however, have both an external format (*i.e.*, the bits sent on the wire) and an internal representation. The internal representation describes data and computations that are not exchanged across the network, but that are essential for parsing and serialisation. While the internal representation is largely implementation-defined, a model of it must be captured in a complete description of the protocol. The techniques described so far couple the external format of the protocol with its internal representation. This is not sufficient for protocols that have a multi-stage parsing process, where persistent state or computations are used. For example, in QUIC, decryption must be modelled to enable parsing.

Recently, extensible **protocol representation systems**, that allow for the description of the external syntax *and* internal representation of protocols, have been introduced. Nail [3] is a tool for generating parsers for data formats. Nail supports multi-stage parsing of complex types using *stream transformations* that take a persistent data store (an *arena*) and an input stream, and output one or more output streams. These can be used, for example, to perform DNS label decompression, highlighting the necessity of providing extensibility beyond a limited set of predefined types. This offers flexibility, but streams and arenas are weakly typed: there is only one stream and one arena type, allowing functions that manipulate these types to be combined in ways that would produce runtime errors.

Approaches are starting to be developed that give stronger type guarantees. Narcissus [8] is a parser combinator-style framework for the Coq proof assistant [9], enabling derivation of verifiable encoders and decoders. Beyond likely usability barriers to the adoption of a framework based on a non-mainstream tool such as Coq, which we turn to in Section 4, Narcissus does not provide support for persistent storage throughout the parsing process.

These approaches show a clear direction, but with some important limitations. The range of domain-specific protocol description languages used by the standards community implies an abstract protocol data modelling language, decoupled from the syntax description, is needed. The type system to model that data also needs to be safe, extensible, and provide support for contextual, multi-stage parsing.

3 NETWORK PACKET REPRESENTATION

Building on the ideas discussed in Section 2, we introduce the *Network Packet Representation*, a typed protocol representation that is independent of the protocol description language.

There are three key innovations in our approach. First, by decoupling the protocol description language from the Network Packet Representation, we allow arbitrary protocol description languages to be used, including ad-hoc, domain-specific languages. We discuss how the Network Protocol

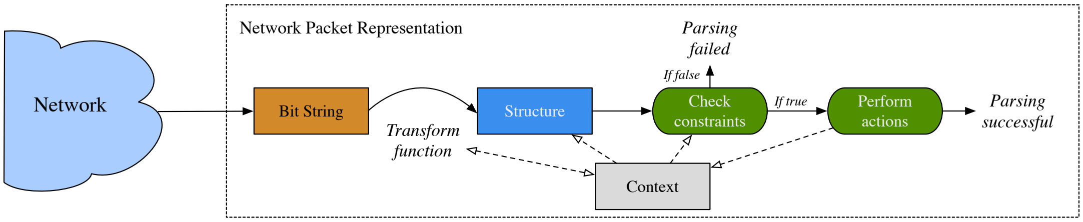

Figure 1: Parsing structure types: *transform functions* instantiate the structure, their *constraints* are checked, and *actions* are performed. The *parsing context* is a persistent data store, accessible throughout the process.

Representation fits within a broader framework in Section 4. Second, we recognise that state must be maintained between PDUs, and to hold out-of-band context, in order to parse many protocols, and provide first class, strongly typed support for a persistent parsing context. Finally, we provide first class support for dependently formatted PDUs, constraints on and between PDU fields, and for multi-stage parsing via typed transformation functions; all of which are needed to parse complex protocols. The Network Packet Representation is a typed description of protocol data. The top-level *Protocol* type describes the overall protocol data, and is parameterised by the basic types representing the protocol data units, the parsing context, and functions.

3.1 Basic Types

The Network Packet Representation includes a number of basic type constructors, that can be used to compose and construct complex protocol descriptions.

Bit string types represent raw protocol data. Distinct bit string types must be defined for each class of protocol data. For example, a protocol that uses both timestamps and source identifiers, each being 32 bits in size, will define distinct bit string types for each, despite them both being identically sized sequences of bits. All protocol data is eventually parsed from, or serialised to, a sequence of bit strings.

Enumerated types represent values that can take different types, dependent on the context. For example, the PDUs of the QUIC transport protocol could be represented as an enumerated type with variants for long header, short header, and version negotiation packets. Enumerated types can also represent substructures within a PDU that can take different forms, such as QUIC frame types or RTCP packet types.

Structure types represent heterogeneous compound data. They represent PDUs, such as TCP segments or QUIC packets. A structure type represents a sequence of fields, each with its own type, packed together in the specified order. Each type has a well-defined size, with no additional padding. Each field has an associated expression that determines if it is present. The expression algebra includes arithmetic, ordinal, equality, and boolean operators. Expressions can also reference the value or length of preceding fields, or of parsing context fields (§3.2). In addition, functions (§3.3) can be used to compute values. For example, RTP data packets [21] include an optional header extension that is dependent on the value of the extension (`X`) field in the fixed header.

Structure types are parameterised by a set of *constraints* on the values of their fields; a set of *actions* that should be executed on successful parsing and that can change state in the parsing context; and the types from which the structure can be created and to which it can be serialised, used to derive *transform functions* that perform parsing and serialisation, as shown in Figure 1.

Constraints determine whether a particular set of fields forms a valid instance of a particular structure type. The same expression algebra is available as for field-level presence expressions; constraints can reference the value of fields within the structure, access the parsing context, and make use of helper functions. A simple constraint could be that a version number field has a fixed value. A more complex example might be an RTP data packet where, if the `P` field in the structure is set to 1, then the `Padding` and `PaddingCount` fields are present, and the length of the `Padding` array is equal to the value stored in `PaddingCount`. Construction of an instance of a structure type fails if any constraint evaluates to `false`. This allows structure types to be variants in an enumerated type, where the variant type is determined by the structure type's constraints.

Finally, structure types must specify types from which they can be parsed, and to which they can be serialised. These are used to derive *transform functions* that indicate how a structure type is constructed from another type, and how it is serialised. In the simplest case, a structure type is constructed from a bit string, and serialises to a bit string. However, there may be intermediary types in more complex scenarios. In QUIC, for example, an unprotected packet is constructed from a protected packet, itself constructed from a bit string. These transformation functions are important: they form part of the parsing process.

Array types represent homogeneous compound data. An array denotes a sequence of values of some other type. Arrays are dependently typed, with an expression determining their length. This may be a constant expression, representing a fixed size array, or may depend on fields of an enclosing type. For example, an RTP packet header contains an array of contributing source identifiers which is sized based on the value of the CC field in the enclosing structure.

3.2 The Parsing Context

PDUs of modern protocols often cannot be parsed in isolation. Rather, parsing depends on contextual information retrieved from previous PDUs or via out-of-band signalling. Examples include: (i) recording the cryptographic keys and other information relating to the security context; (ii) storing parameters specified within earlier packets, to allow for the parsing of later packets; and (iii) storing parameters specified in out-of-band signalling, that are needed to parse packets (e.g., in RTP, signalled payload type values or the meanings of header extensions [22]). The Network Packet Representation maintains a *parsing context* structure to hold this state. This can be read from using the expression algebra used in the definition of types and their constraints, as discussed in Section 3.1. Upon the successful parsing of structure types, the parsing context can be updated. In addition, transform functions have access to the parsing context.

Making the parsing context a fundamental component of the type system is an important distinction from approaches such as Nail [3]. As discussed in Section 2, Nail includes *arenas* that are similar to the parsing context described here, but these are weakly typed. By contrast, each value of the parsing context in the Network Packet Representation is typed; this prevents a large class of runtime errors.

3.3 Functions

Function types can be either *helper functions* or *transform functions*. Helper functions return a value, and are used within the expression algebra. For example, a helper function could compute a checksum, and this value then used as part of an expression. Transform functions are associated with structure types, and define how these types are parsed from and serialised to, as described in Section 3.1.

Importantly, and again in constrast with Nail [3], functions are strongly typed. Transform functions are parameterised by a set of typed parameters, and a return type. This removes a class of runtime errors: types can only be associated with functions that can manipulate them.

4 INTEGRATION WITH STANDARDS

The Network Packet Representation addresses the *technical* challenge of describing protocol data. However, our broader goal is to enable the generation of parser code for a protocol directly from the standards document that describes it. This introduces a number of *social* challenges, requiring standards developers to change how they write documents.

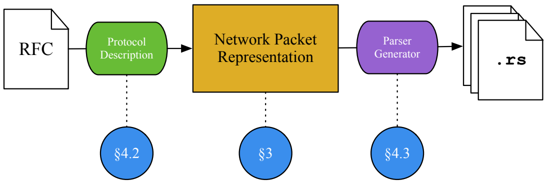

Figure 2: Describing, representing, and parsing PDUs

In this section, we describe a framework for representing the data formats used by a protocol, and for generating parser and serialisation code from this description. As shown in Figure 2, the Network Packet Representation is key to this, providing the intermediate representation in our system. We augment this with a framework for parsing descriptions of protocol data (§4.2), and a code generator that translates the intermediate representation into parser code (§4.3).

4.1 Principles for Standards Integration

We begin by describing a set of broad design principles that are applicable to any approach to describing protocol data that seeks to integrate with the IETF standardisation process:

Most readers are human. As discussed, the standardisation process is rightly centred around prose documents. Standards documents should continue to be written primarily for human readers, who require text and diagrams that they can understand and discuss. Machine readable elements that they cannot easily understand should be avoided.

Authorship workflows are diverse. Documents that include machine readable elements should not require the use of specific tools or workflows. This ensures that disruption to existing workflows, of which there is much diversity, is minimised. The short-term impact of requiring specific tools is that adoption is likely to be slow. In the longer term, those authors with incompatible workflows might be discouraged from participating in the process. This design principle does not preclude the promotion of optional, supplementary tools that aid in the authoring of machine readable elements.

Canonical specifications. Machine readable elements should be part of the canonical specification of the protocol. Replicating elements in both human and machine readable forms within the same document is undesirable, since it creates the potential for inconsistency between the two.

Expressiveness. Machine readable descriptions should not limit expressiveness. However, it is important that specifications remain easy to read and write. An approach that is easy to use, and that addresses *most* use cases, is preferable to a complex approach that addresses *all* cases.

It is desirable that any approach can capture the syntax and parsing steps for most binary protocols. If an approach is not expressive enough then its adoption will be limited, and authors are likely to revert to defining the protocol in prose, making it difficult to parse. However, it may be desirable to limit expressiveness to provide intrinsic safety, security, and computability guarantees. Protocol description languages should be minimally expressive (while accommodating the other principles described), and restrict protocol designs to those for which safe and secure parsers can be generated.

Minimise required change. There are few components of standards documents that aren’t optional. It is not possible to force adoption of a particular approach. The number of changes required to the way that documents are formatted, authored, and published should be minimised.

These principles are not inviolable. The purpose of describing principles is to set out the potential consequences of doing so, and to give guidance rather than to set strict rules.

4.2 Protocol Description Languages

The standards community uses a number of different formats to describe protocol data units. These include formally specified protocol description languages, such as ABNF [7], ASN.1 [23], and YANG [4], semi-formal descriptions such as the TLS presentation language [17, 24], and informal packet header diagrams. As discussed in Section 4.1, any tool that aims to parse protocol data formats from a broad set of standards documents must therefore accept multiple description formats. The Network Packet Representation (§3) is developed to help meet this goal by providing a language agnostic framework for describing protocol data.

Parsing structured protocol description languages is well understood. ABNF and ASN.1 parsers, for example, have long existed. To the extent that the Network Packet Representation can describe formats that are documented in other formal languages, translation should be straightforward.

More challenging is parsing informally specified packet header diagrams. A number of different variants are in wide use, and many documents use inconsistent formatting within their descriptions. Despite this, many of these diagrams are machine readable, and the format can be regularised with only minimal changes. We have developed a parser for such formats that can take as input an IETF RFC or Internet-Draft, and produce a Network Packet Representation of the protocol data units. To support this, we document the Augmented Packet Header Diagram language [14], a variant of commonly used packet diagrams. The format is extremely close to that in common use, so can be adopted by standards developers with no additional training or tooling changes.

A full description of the Augmented Packet Header Diagram language is omitted owing to space constraints. The format remains extremely close to that in common use, so that it can be adopted by standards developers with no additional training or tooling changes. In addition, it balances structure and uniformity, desirable for machine parsing, with the flexibility needed for practical use.

```
An Initial Packet 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
 +-+-+-+-+-+-+-+-+
 |1|1| T | R | P |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                            Version                            |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |   DCID Len    |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |               Destination Connection ID (DCID)              ...
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |   SCID Len    |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                 Source Connection ID (SCID)                 ...
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                         Token Length                        ...
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                            Token                            ...
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                           Length                            ...
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                         Packet Number                       ...
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                            Payload                          ...
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

where:

Header Form (HF): 1 bit; HF == 1.  The most significant bit (0x80) of
    byte 0 (the first byte) is set to 1 for long header packets.

                                 ...

DCID Len (DLen): 1 byte; DLen <= 20.  This field contains the length,
    in bytes, of the Destination Connection ID field that follows it.

Destination Connection ID (DCID): DLen bytes. The Destination
    Connection ID field is between 0 and 20 bytes in length.  On
    receipt, the value of DCID is stored as Initial DCID.

SCID Len (SLen): ...
```

(a) Describing a QUIC Initial Packet

```
A Protected Packet is either a Protected Long Header Packet or a
Protected Short Header Packet.

An Unprotected Packet is either a Long Header Packet or a Short
Header Packet.

An Unprotected Packet is parsed from a Protected Packet using
the remove_protection function. The remove_protection function
is defined as:

func remove_protection(from: Protected Packet) -> Unprotected Packet:
   ...

An Unprotected Packet is serialised to a Protected Packet using
the apply_protection function. The apply_protection function is
defined as:

func apply_protection(to: Unprotected Packet) -> Protected Packet:
   ...
```

(b) Translating between protected and unprotected packets

Figure 3: Describing QUIC with the Augmented Packet Header Diagram language; ‘...’ indicates elided text

We illustrate this, in Figure 3, using an excerpt from a description of the QUIC transport protocol written using the

Augmented Packet Header Diagram language [15]. This describes QUIC Initial packets (Figure 3a) and Protected Packets (Figure 3b). We make a number of observations:

(1) The format is closely similar to existing packet header diagrams. We opt for complexity in parsing the description language, rather requiring change in the process, to ease adoption.
(2) Constraints and the expression algebra are shown: in Figure 3a, the `HF` field must equal 0 if a packet is to be successfully parsed as a long header *Initial* packet.
(3) In Figure 3a, the parsing context is referenced, to hold state that is needed to parse later packets in the flow. Again, we accept parsing complexity to support familiar syntax, with the phrase *On receipt, the value of DCID is stored as Initial DCID* being parsed to add a field, *Initial DCID*, to the parsing context and to add an *action* to the *Initial Packet* structure, to update the context field on receipt of an Initial packet.
(4) In Figure 3b, the `remove_protection` function is defined and linked into the parsing flow.

The result is a minor delta on widely used packet diagrams, that is *both* machine and human readable.

4.3 Parser Generators

Once the Network Packet Representation has been produced it can be used to generate code for an implementation. Our focus, and that of our prototype (http://dx.doi.org/10.5525/gla.researchdata.1025), is on parser generation, but protocol serialisation code can be similarly generated. The use of a common intermediate representation allows us to decouple code generation for the target programming language from the protocol description format. This gives two benefits.

First, a single parser generator can be used, irrespective of the protocol description format. This enables integration with the standards development processes since it doesn't require changes to the way documents are authored to support automatic generation of code from specifications.

Second, commonality in *how* parser generators are created means core functions of code generation can be implemented once, allowing new code generating back-ends to be written with relative ease. This includes type checking the Network Packet Representation, evaluating constraints, and generating stub functions and actions.

Similarly, the order in which types need to be defined is the same across many target programming languages. A depth-first traversal of the intermediate representation type graph enables construction of parser combinators, a common idiom. This traversal begins with those types defined as PDUs, with code for the types contained within composite types generated before the code for the containing type itself. This logic does not need to be reimplemented for each target programming language, easing development.

The result has important implications for security. Protocol description languages can restrict formats to those for which safe and secure parsers can be generated or, at a minimum, ensure that protocol designers are aware of computability and decidability implications [20]. The language-theoretic security (LangSec) paradigm demonstrates further benefits in treating protocols as formal input languages [19].

In practical terms, systems programming languages with an emphasis on memory safety and strict typing are particularly useful for testing the security of new protocols [5]. For this reason, our prototype generates Rust code, with parsers using the `nom` parser combinator library [6].

Considering again Figure 3, our code generator emits types and parser combinator functions to for each field of the packet in turn, followed by the `Initial` packet as a whole. This function returns an `InitialPacket` struct, with its respective fields being instantiated through values returned by the relevant combinator functions. Field and structure constraints are applied for each function call. Stubs are generated for the `remove_protection` and `apply_protection` functions: the function bodies are not captured by the Network Packet Representation, and so must be defined by the programmer. The result can be used as part of a larger implementation.

5 CONCLUSIONS

We have presented a typed representation system that describes the format, parsing, and serialisation of protocols. Our work makes three important contributions. First, the type system is decoupled from the protocol description language, allowing flexibility to overcome the social barriers to adoption. Second, we recognise that state must be maintained between the parsing of individual protocol elements, and support a strongly typed parsing context. Finally, we provide first class support for dependently formatted PDUs, constraints between and within PDUs, and support for multistage parsing; required to parse complex protocols.

In addition, we described a framework and principles for the use of formal description techniques in IETF documents. Our approach is cognisant of the existing process, starting with a familiar input language. This provides an initial, but important, step towards the routine use of tools that can parse documents, bringing a myriad of benefits, and strengthening the trustworthiness of the standards themselves.

6 ACKNOWLEDGEMENTS

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

REFERENCES

[1] G. Back. 2002. DataScript-A specification and scripting language for binary data. In *International Conference on Generative Programming and Component Engineering*. Springer, Pittsburgh, PA, USA, 66–77.

[2] M. Baldi and F. Risso. 2005. A framework for rapid development and portable execution of packet-handling applications. In *Fifth IEEE International Symposium on Signal Processing and Information Technology*. IEEE, 233–238.

[3] J. Bangert and N. Zeldovich. 2014. Nail: A practical tool for parsing and generating data formats. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 615–628.

[4] M. Bjorklund. 2010. YANG – A Data Modeling Language for the Network Configuration Protocol (NETCONF). Internet Engineering Task Force. RFC 6020.

[5] P. Chifflier and G. Couprie. 2017. Writing parsers like it is 2017. In *2017 IEEE Security and Privacy Workshops (SPW)*. IEEE, San Jose, CA, USA, 80–92.

[6] G. Couprie. 2015. Nom, A Byte oriented, streaming, Zero copy, Parser Combinators Library in Rust. 142–148. https://doi.org/10.1109/SPW.2015.31

[7] D. Crocker. 2008. Augmented BNF for Syntax Specifications: ABNF. Internet Engineering Task Force. RFC 5234.

[8] B. Delaware, S. Suriyakarn, C. Pit-Claudel, Q. Ye, and A. Chlipala. 2019. Narcissus: correct-by-construction derivation of decoders and encoders from binary formats. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–29.

[9] The Coq development team. 2020. The Coq Proof Assistant. https://coq.inria.fr.

[10] K. Fisher and R. Gruber. 2005. PADS: a domain-specific language for processing ad hoc data. In *Proc. PLDI*. ACM, Chicago, IL, USA, 295–304.

[11] J. Iyengar and M. Thomson. 2020. QUIC: A UDP-Based Multiplexed and Secure Transport. draft-ietf-quic-transport-latest.

[12] A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan. 2007. Melange: Creating a "Functional" Internet. In *Proc. EuroSys*. ACM, Lisbon, Portugal, 101–114.

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

[14] S. McQuistin, V. Band, D. Jacob, and C. S. Perkins. 2020. Describing Protocol Data Units with Augmented Packet Header Diagrams. draft-mcquistin-augmented-ascii-diagrams-05.

[15] S. McQuistin, V. Band, D. Jacob, and C. S. Perkins. 2020. Describing QUIC's Protocol Data Units with Augmented Packet Header Diagrams. draft-mcquistin-quic-augmented-diagrams-01.

[16] ITU-T Recommendation X.680 (08/15). 2015. Abstract Syntax Notation One (ASN.1): Specification of basic notation.

[17] E. Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. Internet Engineering Task Force.

[18] F. Risso and M. Baldi. 2006. NetPDL: an extensible XML-based language for packet header description. *Computer Networks* 50, 5 (2006), 688–706.

[19] L. Sassaman, M. L. Patterson, S. Bratus, and M. E. Locasto. 2013. Security applications of formal language theory. *IEEE Systems Journal* 7, 3 (2013), 489–500.

[20] L. Sassaman, M. L. Patterson, S. Bratus, and A. Shubina. 2011. The halting problems of network stack insecurity. *USENIX; login* 36, 6 (2011), 22–32.

[21] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. 2003. RTP: A Transport Protocol for Real-Time Applications. Internet Engineering Task Force. RFC 3550.

[22] D. Singer and H. Desineni. 2008. A General Mechanism for RTP Header Extensions. Internet Engineering Task Force. RFC 5285.

[23] International Telecommunication Union. 2015. Abstract Syntax Notation One (ASN.1): Specification of basic notation. ITU-T Recommendation X.680.

[24] A. Walz and A. Sikora. 2017. eTPL: An enhanced version of the TLS presentation language suitable for automated parser generation. In *9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, Vol. 2. IEEE, 810–814.

[25] A. Walz and A. Sikora. 2017. Exploiting dissent: Towards fuzzing-based differential black box testing of TLS implementations. *IEEE Transactions on Dependable and Secure Computing* (2017).