Describing and Parsing QUIC Packets

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

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

ABSTRACT

Standards documents have been slow to adopt packet description formalisms that go beyond the limited syntax that can be captured in ASCII packet diagrams. As increasingly complex protocols, including QUIC, are standardised, this is becoming problematic: descriptions of packet formats are fragmented, incomplete, and difficult to follow, and buggy implementations, that do not conform to the specification, are likely to result. We develop the Network Packet Representation, an abstract, intermediate representation that fully captures the syntax of packet formats, can be generated from a range of packet description languages, and can generate packet parsers. By specifying an intermediate representation that can be produced from both existing and novel packet descriptions we hope to encourage use of more expressive, machine readable, packet descriptions in protocol standards. We describe how the Network Packet Representation can be applied to the definition of QUIC packets – with the eventual goal of allowing QUIC parsers to be automatically generated from the specification.

ACM Reference Format:
Stephen McQuistin and Colin Perkins. 2018. Describing and Parsing QUIC Packets. In *Proceedings of CoNEXT '18*. ACM, New York, NY, USA, 7 pages. https://doi.org/TBA

1 INTRODUCTION

ASCII packet header diagrams are a common formalism found in IETF standards. These allow visualisation of packet formats, reducing human error and aiding implementation of protocol parsers. However, while such ASCII diagrams can capture much of the syntax of existing protocols, like TCP and UDP, the complexity introduced by new protocols, such as QUIC [6], diminishes their utility. Correct parser implementations for such protocol are much more reliant upon the careful interpretation of prose descriptions of the protocol's syntax. This is a situation that is likely to result in buggy and non-conforming parser implementations.

Two broad issues have affected previous efforts to develop formalisms that better capture the syntax of protocols. Firstly, most are largely unsuitable for protocols, like QUIC, that use encryption primitives throughout, including protected payloads. This means that decryption must be modelled. Prior work, which has largely focussed on the parsing of individual unencrypted packets, is unable to model this. As a result, QUIC, and the broader shift towards pervasive encryption [4], require a new formalism.

The second issue with existing approaches is that they generally define a new language that standards authors must use to specify their protocols. This limits their adoption: different documents are suited to different formalisms. As a result, any approach that is likely to be adopted must be flexible in the input formats that can be used.

In this paper, we describe a framework for formalising the description of packet formats. Our framework is comprised of three components: an input parser, a type system and intermediate representation (the Network Packet Representation), and an output formatter. This architecture allows packets to be described using a number of different formats, and for these descriptions to be parsed into a common intermediate form. Output formatters can then use this intermediate representation to generate various artefacts, including parser code. We provide an example of how this framework can be used to model the complexity of parsing QUIC.

There have been many previous attempts at defining a packet format definition language, including PacketTypes [8], Melange [7], PADS [5], and DataScript [1]. These languages typically focus on the parsing of individual, unencrypted packets. Ad-hoc languages, such as the presentation language used in the TLS 1.3 specification [10], also see limited adoption. In contrast to these language definitions, we develop an abstract protocol representation that, with *contexts* and *functions* can describe the parsing of any protocol, and can be generated from existing formalisms.

We structure the remainder of this paper as follows. Section 2 expands on our motivation, by assessing the utility provided by an ASCII diagram from the QUIC standards documents, and outlining the requirements of our formalism.

CoNEXT '18, December 4–7, 2018, Heraklion/Crete, Greece

ACM ISBN TBA…$15.00
https://doi.org/TBA

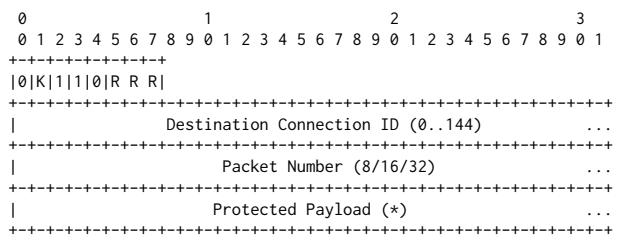

Figure 1: QUIC short header format (from [6])

Section 3 describes our proposed framework for describing packet formats, including our intermediary representation, the Network Packet Representation. Section 4 provides an example of how our framework can be used to capture the parsing of QUIC's protocol data units (PDUs). Finally, Section 5 discusses related work, and Section 6 concludes.

2 MOTIVATION & REQUIREMENTS

Broadly, the requirements for our formalism are that it should:

- be able to fully capture the information required for *parsing* PDUs;
- accommodate various PDU specification languages, including ASCII packet header diagrams, ad-hoc languages such as the TLS 1.3 presentation formation, and more generic packet description languages;
- produce output in various formats, including ASCII diagrams and parser code.

To demonstrate the limitations of ASCII packet header diagrams in capturing all of the information required to fully parse PDUs, we consider the diagram for QUIC's short header PDU, as shown in Figure 1. It is clear how the first octet should be parsed: the widths of each field are known, and, for some, the values specified. However, it is not clear how the Destination Connection ID, Packet Number, and Protected Payload fields should be parsed. Determining the syntax of these fields requires the accompanying prose descriptions: the expressiveness of ASCII packet diagrams is limited.

Some of these limitations can be overcome by using an existing packet description language. Such languages typically allow for intricate field encodings to be specified. For example, packet numbers use a variable-length integer encoding format, where the most significant bits of the first octet determine the length of the field. While it is impractical to represent this behaviour in an ASCII diagram, most packet description languages enable this format to be captured.

However, we have identified two broad features that are missing from existing packet description languages that mean that they cannot capture the behaviour of all protocols, including QUIC, where encryption is part of the parsing process, or RTP, where PDUs cannot be parsed independently.

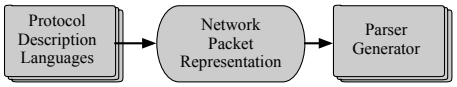

Figure 2: Describing, representing, and parsing PDUs

Firstly, existing languages typically enable per-packet modelling, but decryption in QUIC requires information from packets earlier in the flow. We require flow-level *context*: essentially a key-value store that can be added to as parsing progresses, and modelled as part of the protocol specification. For example, the nonce for decrypting the protected payload field is derived from the packet number. However, the packet only contains the least significant bits of this number. Contextual information from other packets in the flow is required to generate the nonce. As an additional example where contextual information is required for parsing, we consider RTP [11]. RTP includes support for header extensions, whose format is signalled out-of-band using SDP; context is needed to share information between different parsing events, and out-of-band processes.

The second feature needed to fully model the parsing of QUIC is the ability to capture *how* encrypted fields are parsed. Existing description languages have been able to treat encrypted fields as unparsable blobs, enabled by the separation between protocols. For example, where TCP is used to carry an encrypted payload, the format of the payload is not specified as part of TCP: the parsing of TCP can end with the payload as an encrypted blob. However, in QUIC, parsing a short header packet requires the protected payload by decrypted, and further parsed into frames. As a result, we must model the *functions* needed to parse of the protocol.

To fulfil our second two requirements – flexibility in both input and output formats – our formalism must be an abstract, intermediate representation language. Such a representation can be generated from an arbitrary set of input formats, and can itself be used to produce various outputs. This allows for existing formalisms, including ASCII packet diagrams and the TLS 1.3 presentation language, to be used. The expressivity of the intermediate representation depends on the input format; this allows for adoption while incentivising more expressive formalisms.

3 DESCRIBING PROTOCOL DATA

We introduce the Network Packet Representation, a type system the allows the format of protocol data units to be specified, alongside information on how those PDUs should be parsed. Key to this is the definition of a type system (§3.1) that describes the format of PDUs, helper functions, and the context needed for parsing.

This is associated with one or more protocol description languages (§3.2) that describe the PDUs to be parsed; and

one or more parser generators (§3.3) that can generate implementations of parsers for those PDUs. These components are illustrated in Figure 2.

3.1 A Type System for Protocol Data

The Network Packet Representation type system describes the format of PDUs, along with the helper functions and context needed to parse them.

The base type class used to describe PDUs is a *Bit String*. This is a sequence of *n* consecutive bits of data that can be instantiated as a named type. For example, a type system to describe RTP [11] might instantiate a new instance of the bit string type class, `SeqNum`, comprising 16 bits to represent an RTP sequence number.

There are three classes of derived types that represent structured data in a PDU: array types, structure types, and enumerated types. An *Array Type* represents a sequence of values of some other type; it may have fixed or unspecified length. If the length is unspecified in the definition of the array type, it must be possible to infer it based on the constraints of an enclosing structure type, discussed below.

A *Structure Type* represents a heterogeneous collection of fields. They are the main types used to represent packet formats. At a high-level, structure type definitions resemble those in languages such as C: they comprise a sequence of fields, each with its own type, that are packed together in the order specified. Unlike C, no padding can be present between fields, and the size of all types is well-defined and derived from the size of the underlying bit string types (there is no undefined or implementation defined behaviour).

Structure types represent protocol data that can be parsed. For example, a QUIC frame, or a TCP header. The format of a particular instance of a structure type, as it is parsed, is determined by a set of constraints on the fields that structure, and transformations of those fields. There are two types of *Constraint Expression* that can affect parsing of a structure instance: *is_present* constraints on specific fields, and structure-level constraints. The former are attached to optional fields, and are boolean expressions that declaratively indicates whether those fields are present when the structure is parsed. The latter are attached to the structure as a whole, and specify more complex constraints, potentially across multiple fields, that must hold for the packet to be valid when parsed. Constraint expressions can reference the values of fields of the enclosing structure, methods defined on those fields, fields in the parsing context, or helper functions. For example, in the QUIC short header (shown in Figure 1), the first bit is defined as the *header type* field, whose value is constrained to being equal to 0.

Enumerated Types allow specification of alternatives: a PDU can be either s PDU of type *X* or type *Y*.

The *Parsing Context* allows the parser to keep state between packets. It is essential to parsing complex protocols, and the ability to specify data to be stored in the context is one of the key innovations of our Network Packet Representation. Typical uses of the parsing context include: 1) recording the cryptographic keys and other information relating to the security context; 2) storing parameters set in early packets, to allow parsing of later packets; and 3) 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 [12]). Structure types can also be tagged with a series of *Actions* that are performed when they are parsed; these are generally expressed in terms of calls to helper functions that update the parsing context.

The Network Packet Representation allows the definitions of *Function Prototypes*. These can be either *Helper Functions* or *Transform Functions*. The function prototypes specify the function name, parameter names and types, and the return type only; they do not specify the body – the Network Packet Representation is a type system not a complete programming language. Helper functions specify constraints or actions; transform functions define how a field is processed after parsing to generate a replacement field. A common example of the latter might be a `decrypt()` function that takes a key previously stored in the parsing context (e.g., by a call to an action function specified on parsing of an initial handshake packet) to decrypt a protected payload field. The intent is that the semantics of the function are specified in prose, in the protocol specification, while its arguments and return type are specified in the input protocol description language (see Section 3.2). Together, constraint expressions and transform functions allow structure types to represent complex PDU formats, such as those used in QUIC.

Finally, the *Protocol Type* includes definitions for the other types, and gives the types of the PDUs used in the protocol.

The type system of the Network Packet Representation is supported by an *intermediate representation*; a well-defined serialisation format that can be generated from the front-end protocol description languages and used to instantiate the types describing the protocol data.

3.2 Protocol Description Languages

The intermediate representation that is used to construct the Network Packet Representation is generated by programs that parse a protocol description language, rather than by standards authors. This decoupling means that *any* protocol description language can be used, including ad-hoc, domain-specific languages. This is in contrast to previous approaches that mandated a common description language.

The flexibility of this approach means that the definition of a protocol description language can be broad, ranging

from ASCII packet diagrams, to the TLS 1.3 presentation language. Importantly, these languages are already in use: parsers can be written for them to gain the benefits of having a common representation. As we describe in Section 5, many description languages have been already been defined; these too can be used to generate the intermediate representation.

Building a parser for an existing language allows for widespread adoption of the Network Packet Representation, and enables the benefits that come with having a common representation. However, a key motivation for the representation is that existing languages cannot model features of common protocols. The expressiveness of the language used to generate the intermediate representation determines how well the protocol is defined. For example, ASCII packet diagrams do not allow for relationships between fields to be defined; these would then be missing from the intermediate representation. As a result, the use of existing languages can be viewed as a transitional phase: as the benefits of a common representation become clearer, more expressive languages can be adopted as part of the standardisation process. We use one such language as part of our QUIC example in Section 4.

3.3 Parser Generators

Once the intermediate representation has been produced, it can be used to generate a parser for the protocol that it describes. Again, the flexibility of having a common representation means that a parser can be generated for *any* target language, such as C or Rust. Previous approaches would have required writing a parser generator for each protocol description language: decoupling these components means that only one parser generator is needed for each target language.

Commonality in *how* parser generators are created means that tooling can be used to support their development. For example, in many target languages, the order in which types need to be defined is the same. Tooling could implement this logic once, easing the development of parser generators.

There are two main benefits that are derived from generating parsers from a common intermediate representation. Firstly, the parsers that are generated will be compliant with the described protocol, and are less likely to introduce bugs. In addition, once a parser generator has been written, parsers can be generated in novel, safe languages, such as Rust [3], without additional effort.

4 CASE STUDY: PARSING QUIC

In this section, we will demonstrate that the novel features of the Network Packet Representation – helper functions and context – are necessary in describing QUIC. We illustrate this using an example that formalises some of QUIC's PDUs using a protocol description language. As discussed in Section 3, the description language itself is not important:

```
VarInt := {
        two_bit       : Bit2;
        value         : Bits;
} where {
        value.length == (8*(2^two_bit))-2;
};

PacketNumber7 := {
        first_bit     : Bit;
        packet_number : Bit7;
} where {
        first_bit == 0;
};

PacketNumber14 := {
        first_twobits : Bit2;
        packet_number : Bit14;
} where {
        first_twobits == 1;
};

PacketNumber30 := {
        first_twobits : Bit2;
        packet_number : Bit30;
} where {
        first_twobits == 2;
};

PacketNumber := { PacketNumber8 | PacketNumber16 | PacketNumber32 };

Context := {
        highest_packet_num  : PacketNum;
        last_scid           : ConnectionID;
};
```

Figure 3: Derived data types for QUIC parsing

```
lh_packetnum_decrypt :: (ppacket_num : Bits,
                         context : Context)
                      -> PacketNum;

LongHeader := {
        header_form   : Bit;
        packet_type   : Bit7;
        version       : Bit32;
        dcil          : Bit4;
        scil          : Bit4;
        dcid          : Bits;
        scid          : Bits;
        length        : VarInt;
        ppacket_num   : Bits -> packet_num : PacketNum;
        payload       : Frame[];
} where {
        header_form == 1;
        dcid.size == ((dcil == 0) ? 0 : dcil+3);
        scid.size == ((scil == 0) ? 0 : scil+3);
} onparse {
        packet_num = lh_packetnum_decrypt(ppacket_num, Context);
        Context.packet_num = packet_num;
};
```

Figure 4: QUIC long header (from [6])

the Network Packet Representation is agnostic to input and output formats. The syntax of the language we use here has been designed to demonstrate the functionality provided by the type system. In addition, examples used in this section are not exhaustive: we look at the long and short header packet formats, and how their parsing can be modelled. We

do not, for example, consider packet-level protection (e.g., 0-RTT protection for long header packets), but this does not limit the generality of our approach.

We begin by describing QUIC's long header format, as shown Figure 4. This code declares a structure type named `LongHeader` with the fields specified in the first block. The `Bit` and `BitN` types used in this definition are instances of the bit string type, with sizes 1 and N respectively; the `Bits` type is a bit string with an undefined length. The derived data types used in the `LongHeader` structure are defined in Figure 3. The `VarInt` type is a structure that encapsulates QUIC's variable-length integer encoding format. `PacketNum` encodes the similar variable-length packet number format as an enumerated type, where the alternatives depend on the first one or two bits of the format to determine the width of the packet number.

The `where` block further defines the `LongHeader` type, by constraining the values that the structure's fields can take. A PDU is a `LongHeader` type only if all of the constraints in the `where` block are true. This highlights the benefit of this programmatic description of the long header: the relationship between the `dcil`/`scil` and `dcid`/`scid` fields is made explicit. In the standards documents, where ASCII packet diagrams are used, the `dcid`/`scid` fields are shown as having variable lengths; the prose description accompanying the diagram is needed to fully describe the packet format.

The `onparse` block contains actions that are to be performed once the PDU has been successfully parsed as a `LongHeader` type (i.e., its layout matches that of the structure, and all of the constraints in the `where` block are true).

The single expression in the `onparse` is matched with the following from the definition of the structure:

```
ppacket_num : PPacketNum -> packet_num : PacketNum
```

This is a field transformation: it indicates that the parsed block contains a protected packet number field (with type `Bits`), which, to allow for further parsing, is then transformed (using the `lh_packetnum_decrypt` function) into a packet number field (with type `PacketNum`). The intermediate representation captures the signature of the functions required in the parsing of the protocol. At present, much of this behaviour is captured in prose descriptions of the protocol, limiting the usefulness of the ASCII packet diagrams the documents provide. This issue is exacerbated by the TLS-related functionality being described in a separate document (e.g., packet number protection is described in Section 5.3 of the QUIC-TLS draft [14]). Using function names would explicitly link these components together.

Finally, Figure 4 makes use of the `Context` type, defined in Figure 3. As described in the previous section, the parsing context is essentially a key-value store. Type declarations can access the context, to check against values that it contains, and methods, helper functions, and out-of-band processes

```
decrypt :: (ppayload : Bits,
            context : Context)
        -> Frame[];

sh_packetnum_decrypt :: (ppacket_num : Bits,
                         context : Context)
                     -> PacketNum;

ShortHeader := {
        header_form   : Bit;
        key_phase     : Bit;
        third_bit     : Bit;
        forth_bit     : Bit;
        demux_bit     : Bit;
        reserved      : Bit[3];
        dcid          : Bits;
        ppacket_num   : Bits -> packet_num : PacketNum;
        ppayload      : Bits -> payload : Frame[];
} where {
        header_form == 0;
        third_bit == 1;
        forth_bit == 1;
        demux_bit == 0;
        dcid == Context.scid;
} onparse {
        packet_num = sh_packetnum_decrypt(ppacket_num, Context);
        Context.packet_num = ((packet_num > Context.packet_num)
                              ? packet_num
                              : Context.packet_num);
        payload = decrypt(ppayload);
};
```

Figure 5: QUIC short header (from [6])

```
PaddingFrame := {
        frame_type   : VarInt;
} where {
        frame_type == 0;
};

Frame := { PaddingFrame | RSTStreamFrame | ConnectionCloseFrame | .. };
```

Figure 6: QUIC frames (from [6])

can access and change values in the context. In this example, the context includes a `highest_packet_num` field. The field transformation captures that the `lh_packetnum_decrypt` function removes the packet number protection, and decodes the truncated packet number into the full packet number. The decoding process requires not only the truncated packet number (i.e., the unprotected incoming packet number), but also the largest packet number seen so far. As shown, this is stored in the context, which the `lh_packetnum_decrypt` function can access.

Next, we consider the short header, as shown in Figure 5. We omit discussion of the features already discussed as part of the long header example. The short header also includes a reference to the context. The short header's `dcid` field must match the most recent source connection ID that has been sent. This demonstrates another way in which the context can be accessed: the sending process is essentially out-of-band, but information must be shared with the parsing process to allow it to progress.

This is the rationale behind helper functions and parsing context: decryption primitives are core to the QUIC protocol. If we are to design an intermediate language that can model the parsing of QUIC PDUs, it *must* allow for the modelling of decryption. This is further illustrated by the link between the short header format's protected payload, and the QUIC frames (as shown in Figure 6) that it ultimately contains. It is not sufficient to treat the protected payload as an encrypted blob: this does not fully model the parsing process. As shown, a short header PDU is parsed, and the payload is decrypted using the `decrypt` function, which takes both the protected payload and the context (for any information needed to decrypt the payload), and returns an array of frames. The frames are then parsed as the `Frame` type defined in Figure 6.

The example given here also hints at the boundary between modelling the parsing of PDUs, and validating that those PDUs make sense within the broader semantics of the protocol. For example, we use enough logic to store the highest packet number seen, but do not otherwise validate incoming packet numbers. The intermediate representation is captures only the information that is required to parse PDUs: ultimately, PDUs might parse successfully, only to be invalid, in terms of the semantics of the protocol.

5 RELATED WORK

There have been many previous efforts to develop protocol description languages. Some, such as PADS (Processing Ad Hoc Data Sources) [5] and DataScript [1], are C-like in their syntax and type systems. Others, like PacketTypes [8] and the Meta Packet Language (MPL) [7], are declarative, with more expressive type systems. These approaches don't support contexts or helper functions: they can't model the decryption primitives that are essential to the parsing of QUIC, or the out-of-band information needed to parse protocols like RTP. While these languages are well-motivated (i.e., parser quality would be improved by formal specifications in standards documents), they misunderstand the needs of standards authors, and provide no way of transitioning from existing approaches, to those that they define.

In terms of adoption, it's interesting to consider approaches that see use in IETF documents. Most widespread is the ASCII packet diagram. While this is limited in its expressiveness, its value in showing the layout of protocol PDUs should not be ignored. Other formalisms, such as ASN.1 [9] and the TLS 1.3 presentation format [10], see widespread adoption within certain IETF working groups, but almost no adoption in most. This points to the problem noted above: a single input language is not suitable in all cases. The flexibility of our approach – defining a common intermediate representation and type system from which parsers can be derived, rather than a single protocol description language that authors must write – means that existing formalisms, e.g., ASCII diagrams and the TLS 1.3 presentation language, can be used as input formats, with other representations, such as that used in Section 4, being developed as they become useful.

The flexibility is also beneficial in terms of the output formats that can be derived. Beyond the traditional parser languages described in Section 3.3, we also consider P4 [2, 13], a language for programming the data plane of network devices. Widely used in software defined networking applications, P4 describes how packets are parsed and serialised and what state tables exist in a network devices, and the packet processing and forwarding behaviour of these devices. Parsers in P4 are implemented imperatively, and are built-up from a series of state machines. In terms of expressive power, parsers in P4 are comparable to the parsers we describe and can parse the same formats. In our approach, we declaratively define the packet format, and derive the parser implementation from that declaration. P4, on the other hand, simply specifies the parser implementation. As such, P4 could be an appropriate output language for our system.

6 CONCLUSIONS AND FUTURE WORK

We have described the Network Packet Representation, a type system that allows the format and parsing of protocol data units to be described. This builds on existing approaches in three important ways. First, in recognising that encryption primitives are a fundamental component of new protocols, we allow for helper functions to be modelled. A related addition is the inclusion of a parsing context, to allow for data to be exchanged between the parsing of different PDUs, or out-of-band logic. Finally, the representation defines an intermediate representation that can be generated from any input format, and that can be used to generate parsers.

By defining a common type system for protocol data, we aim to provide the benefits of formal specifications without mandating change to the way that protocols are currently specified. Our intermediate representation can be generated by parsing ASCII diagrams as they are written today, for example. While this representation would lack the expressiveness of more formal description (such as the examples in Section 4), it would provide some of the benefits, and incentivise authors to move beyond ASCII diagrams to more sophisticated description formats. We outline one such format that might be used to define the syntax of QUIC packets. We believe that this piecemeal approach fits better with the IETF standards process: common infrastructure and tooling, but no mandated input format or output language.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] Godmar 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] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. 2014. P4: programming protocol-independent packet processors. *Computer Communication Review* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

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

[4] S Farrell and H Tschofenig. 2014. Pervasive Monitoring Is an Attack. RFC 7258. (May 2014).

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

[6] Jana Iyengar and Martin Thomson. 2018. QUIC: A UDP-Based Multiplexed and Secure Transport. draft-ietf-quic-transport-latest. (Aug. 2018).

[7] Anil Madhavapeddy, Alex Ho, Tim Deegan, David Scott, and Ripduman Sohan. 2007. Melange: Creating a "Functional" Internet. In *Proc. EuroSys*. ACM, Lisbon, Portugal, 101–114.

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

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

[10] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. draft-ietf-tls-tls13-28. (March 2018).

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

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

[13] The P4 Language Consortium. 2018. $P4_{16}$ Language Specification. Document available online. (May 2018).

[14] M. Thomson and S. Turner. 2018. Using Transport Layer Security (TLS) to Secure QUIC. draft-ietf-quic-tls-14. (Aug. 2018).