Literature Survey: Improving Protocol Standards for a more Trustworthy Internet

Stephen McQuistin Colin Perkins

20th January 2020

Abstract

This document contains notes on the papers read as part of a literature survey for the "*Improving Protocol Standards for a more Trustworthy Internet*" project.

1 Packet format definition languages

One of the objectives of the project is to introduce (relatively simple) formalisms into standards documents, such that tooling can be used to improve the quality of these documents. One area where documents vary in quality, and that could easily be improved by our proposed approach, is in the description of packet header formats. The format used to describe these in standards documents varies, albeit slightly, complicating their extraction and use by automated tools. Further, given that they are a common feature in standards documents, any approach that allows for their specification to be simplified (from the view of the author) is likely to be welcomed.

One approach to improving packet header definitions is through the use of a domain-specific language (DSL). Such a DSL needs to allow the author to specify not only the width and type of each field in the packet header, but also the relationships between them, and any constraints that apply. The language must be accessible: it should reduce the burden on the author, rather than adding additional complexity to the standards process.

The literature survey will cover not only languages specifically designed for packet header definitions, but data format specification languages more generally. In addition, architectural papers (i.e., those that don't define a language, but provide guidance) are also in scope.

1.1 Languages

Paper: "*Packet Types: Abstract Specification of Network Protocol Messages*" – McCann & Chandra [26]

McCann and Chandra propose PacketTypes, a packet specification language. Their motivation is largely focussed on improving the efficiency with which packets can be parsed in systems applications, though they note that the formal specification of packet formats is likely to benefit standards specifications. They note that packet formats are fixed by standards documents, and that implementing parsers is often complicated by the interaction between different fields, and across the different layers. They use the IP header as an example: the payload format is constrained by its own packet format, but this is determined by the protocol field in the IP header. Further, the IP header contains a variable-length options field; the length of this field is determined by the header length field. More generally, they show that packet headers encapsulate sufficient complexity that C's type system is insufficient. As a result, they develop PacketTypes: a specification language for defining packet formats. They provide an implementation for their language, and demonstrate its use by implementing a parser for Q.931 messages.

PacketTypes has one primitive type, a `bit`. New types can be constructed using the syntax `name := type`; for example, `byte := bit[8]` and `bytestring := byte[]`. The square bracket operator (`[]`) is the same as the Kleene star: here, a `bytestring` is comprised of zero or more bytes. Using an integer inside the square brackets restricts the number of repetitions of the type to be exactly that number – so a `byte` is comprised of exactly 8 `bit`s.

Structures (e.g., packet headers) can be formed by grouping type definitions together. From the paper, the IP packet can be defined as:

```
nybble := bit[4];
short := bit[16];
long := bit[32];
ipaddress := byte[4];
ipoptions := bytestring;

IP_PDU := {
```

```
  nybble       version;
  nybble       ihl;
  byte         tos;
  short        totallength;
  short        identification;
  bit          morefrags;
  bit          dontfrag;
  bit          unused;
  bit          frag_off[13];
  byte         ttl;
  byte         protocol;
  short        cksum;
  ipaddress    src;
  ipaddress    dest;
  ipoptions    options;
  bytestring   payload;
}
```

While `IP_PDU` defines the layout of an IP packet, without any constraints on the values that each field can contain, by this definition alone, an invalid IP packet could be constructed. PacketTypes allows a `where` clause to follow a type definition. For example:

```
IP_PDU := {
  ...
} where {
  version#value = 0x04;
  options#numbytes = ihl#value * 4 - 20;
  payload#numbytes = totallength#value - ihl#value * 4;
}
```

In this example, the `where` clause specifies that the IP version is 4, and defines the size of the `options` and `payload` fields based on the `ihl` and `totallength` fields. Note the introduction of new syntax here: `field#attribute` is used to refer to a particular attribute of a field. A number of attributes are defined, including `value` (network-order concatenation of all bits in the field), `numbits` (count of the number of bits in the field), `numbytes` (count of the number of bytes in the field), `numelems` (count of the number of elements in an array field), and `alt` (booleans indicating the alternative chosen). In the absence of a constraints, the `[]` operator is greedy; it is assumed that any data that could belong to that field does. It might be constrained explicitly by the `where` clause, or implicitly, based on the length of the structure that holds it.

A refinement operator, `:>` is defined. This is used to further constrain a type that has already been specified. From the paper, an Ethernet frame could be specified as:

```
macaddr := bit[48];

Ethernet_PDU := {
  macaddr      dest;
  macaddr      src;
  short        type;
  bytestring   payload;
}
```

Using a refinement, an IP packet within an Ethernet frame could be defined as:

```
IPinEthernet :> Ethernet_PDU where {
  type#value = 0x800;
  overlay payload with IP_PDU;
}
```

This specifies the type as `0x800`, and overlays the `IP_PDU` definition above on top of the Ethernet payload. The `overlay .. with` syntax embeds one type within the other, reflecting protocol encapsulation. Here, the constraints of the `IP_PDU` definition are applied to the `payload` field of `Ethernet_PDU`; for a packet to be a valid `IPinEthernet` packet, its Ethernet header must have `type` equal to `0x800`, and a payload that is a valid `IP_PDU` packet.

PacketTypes includes dot notation syntax to allow the fields in the overlaid structure to be accessed:

```
My_IPinEthernet :> IPinEthernet where {
  payload.srcaddr#value = 192.168.0.1;
}
```

In this example, the `payload` is the `Ethernet_PDU` payload; the dot notation allows for the `srcaddr` of the overlaid `IP_PDU` definition to be accessed, and here, have its value specified.

Alternative types can be specified using the alternation operator, `|=`. This combines types disjunctively – a bitstring matches the defined type if it matches one of the alternatives. As an example, the paper gives a more precision definition of the `ipoptions` type:

```
ipoptions := {
  NonEndOption     neo[];
  EndOption        eo;
  bytestring       padding;
} where {
  eo#numelems <= 1;
}
```

By this definition, IP options are comprised of a zero or more `NonEndOption`s, followed by an optional (zero or one) `EndOption`, and padding. `NonEndOption`s make use of the alternation operator:

```
NonEndOption |= {
  NoOperation     nop;
  Security        sec;
  LSRR            lsrr;
  SSRR            ssrr;
  RR              rr;
  StreamID        sid;
  Timestamp       tstamp;
}
```

The `|=` means that a bitstring is a `NonEndOption` if and only if it is one of the alternative types given. The `#alt` attribute can be used in the `where` clause to determine which alternative was matched. Boolean expressions can be constructed; for example `neo[0]#alt @ nop` is true if the first `NonEndOption` in the IP options field is a `NoOperation`. Note the use of the `@` operator here.

Constraints in the `where` clause can be expressed using relational operators and boolean combinators:

```
= > < >= <= != || && + * / -
```

PacketTypes limits, for computability, the possible set of constraints. Constraints that specify the size of variable-sized fields must only reference attributes that have already been specified. The authors allow particular implementations to further limit constraints.

Paper: "*Melange: Creating a "Functional" Internet*" – Madhavapeddy et al. [25]

The motivation for Melange is centres on the widespread use of type-unsafe languages (like C and C++) in network system implementations, exposing such systems to security and reliability issues. Melange combines two techniques to eliminate the perceived performance impact of type-safe languages, allowing for their use in implementing Internet protocols: a strong, static type system, and generative meta-programming. Of interest here is their domain-specific language, Meta Packet Language (MPL).

The authors provide the Extended BNF grammar for MPL:

```
main -> (packet-decl)+ EOF
packet-decl -> packet identifier[(packet-args)] packet-body
packet-args -> {int | bool} identifier [, packet-args]
packet-body -> {(statement)+}
statement -> identifier : identifier [var-size] (var-attr)*;
           | classify (identifier) {(classify-match)+};
           | identifier : array(expr) {(statement)+};
           | ();
classify-match -> '|' expr : expr[when (expr)] -> (statement)+
var-attr -> variant {('|' expr {->|=>} cap-identifier)+}
          | {min|max|align|value|const|default} (expr)
var-size -> [expr]
```

```
expr -> integer | string | identifier | (expr)
      | expr {+|-|*|/|and|or} expr
      | {-|+|not} expr
      | true | false
      | expr {>|>=|<|<=|=|..} expr
      | {sizeof|array_length|offset} (expr-arg)
      | remaining()
```

A simple specification in MPL consists of an ordered list of named *typed* fields. Types can be: (i) wire types (network representation of the field); (ii) MPL types (used within the specification – strings in the grammar above); or (iii) language types (used in target language – i.e., the language that the MPL is compiled to). Wire types are mapped to MPL types, allowing these types to be used as part of the specification. Additionally, every wire type has a corresponding language type.

The grammar includes the `classify` keyword, to allow for parsing decisions to depend on the contents of another field (so long as that field has already been defined). Fields have attributes; these can be constraints that define the maximum or minimum value, a default value, or a constant value, among others.

As an example, the authors provide an MPL specification of IPv4:

```
packet ipv4 {
  version: bit[4] const(4);
  ihl: bit[4] min(5) value(offset(options) / 4);
  tos_precedence: bit[3] variant {
    |0 => Routine
    |1 -> Priority
    |2 -> Immediate
    |3 -> Flash
    |4 -> Flash_override
    |5 -> ECP
    |6 -> Inet_control
    |7 -> Net.control
  };
  delay: bit[1] default(false);
  throughput: bit[1] default(false);
  reliability: bit[1] default(false);
  reserved: bit[2] const(0);
  length: uint16 value(offset(data));
  id: uint16;
  reserved: bit[1] const(0);
  dont_fragment: bit[1] default(0);
  can_fragment: bit[1] default(0);
  frag_off: bit[13] default(0);
  ttl: byte;
  protocol: byte variant {
   |1 -> ICMP
   |2 -> IGMP
   |6 -> TCP
   |17 -> UDP
  };
  checksum: uint16 default(0);
  src: uint32;
  dest: uint32;
  options: byte[(ihl * 4) - offset(dest)] align(32);
  header_end: label;
  data: byte[length-(ihl*4)];
}
```

Paper: "*PADS: A Domain-Specific Language for Processing Ad Hoc Data*" – Fisher and Gruber [18]

Processing Ad hoc Data Sources (PADS) is a declarative data description language for describing the layout and semantic properties of *ad hoc* data. Ad hoc, in this context, refers to data formats that are stored in (previously) ill-defined, non-standard formats. This in contrast to data stored in database systems, or described using a structured language (e.g., XML). The authors cite examples of such data: call logs, web server access logs, etc. While much of the motivation for PADS does not apply here (i.e., we

are not describing ad hoc data), of interest is the motivation that runs through most of the papers in this document: parsers are tedious to write, and the process is error-prone. A formal or structured description of the data is desirable to improve this situation.

The PADS description language is typed-based, with atomic base types for 32-bit integers (`Pint32`), strings (`Pstring`), and IP address (`Pip`), among others. These types are augmented with information about how the data is coded (e.g., in ASCII or binary). PADS interprets the types as ASCII by default, but this can be specified by using the appropriate base type: for example, `Pa_int32` is an *ASCII* 32-bit integer, `Pb_int32` is a *binary* 32-bit integer. Users can specify their own base types.

PADS includes structured types, akin to C's types: `Pstruct` (structures), `Punion` (alternatives), and `Parray` (sequences), `Penum` (fixed set of literals), `Popts` (optional data). Each of these can have a predicate associated with it, to indicate whether a parsed value is a legal value for the type. For example, a predicate might specify that two fields of a `Pstruct` are related, or that a sequence is in increasing order. These predicates are specified using C-like syntax. Further constraints can be added to existing types using `Ptypedef`. PADS types may be parameterised by values. For example, `Pstring(:' ':)` defines a string terminated by a space.

Paper: "*DataScript: A Specification and Scripting Language for Binary Data*" – Back [1]

Back describes DataScript, a language for the specification and manipulation of binary data formats, using types. It consists of two components: a constraint-based specification language (that uses DataScript types), and a language binding that provides a simple scripting interface. The motivation for DataScript is that natural language prose is not well suited to describing the design of binary data formats: not only is it inefficient, but it can introduce ambiguities that translate into bugs in implementations.

A DataScript type can be either: (i) a primitive type; (ii) a set type (an enumerated or bitmask type); (iii) a linear array (of another type); or (iv) a composite (record or variant-record) type.

Primitive types form the basis for more complicated types. These types include bit fields, bytes, and variable length integers. Primitive types are interpreted as signed or unsigned integers, with the size and signedness specified in the keyword (e.g., uint32, int16). For multi-byte integers, endianness is specified using an optional attribute prefix: `little` or `big`. Where this isn't specified, the default is big endian (i.e., network byte order).

Two set types are supported: enumerated types (`enum`) and bitmask types (`bitmask`). A set type constrains an underlying primitive type, specifying the signedness and endianness, used to store and interpret its values. For example, it can be used to specify the valid values of a bitfield.

Composite types are specified using a C-like language, and this follows from the interpretation of these types: they are similar to C structs and unions. Unions in DataScript are discriminated/tagged, taking the first matching constraint in the union – the order in the specification is important. The constraints don't need to be disjoint, allowing for default choices. Composite types can be nested, with scoping rules that provide a namespace for each type.

DataScript arrays are linear, with integer indices, and specified lower and upper bounds (`[lower.. upper]`; lower is a valid indice, upper is not); lower, if omitted defaults to 0. Expressions specifying bounds must be able to be evaluated at the time that the array is read (i.e., be based on a previously defined field).

Fields can have constraints, specified as a boolean predicate. Constraints are used in three ways: (i) to discriminate union types; (ii) express consistency requirements in record types; and (ii) limit the length of array types, where the length is not known; arrays can grow until their constraints are violated. Predicates are boolean expressions, with DataScript adopting the operators, associativity, and precedence rules from Java and C.

Paper: "*P4: Programming protocol-independent packet processors*" – Bosshart et al. [11]

P4 is a high-level language, designed for programming packet processors (e.g., switches). The goals of P4 are to allow for easy reconfigurability of switches once deployed, the development of an abstraction to allow switches to be protocol agnostic, and a further abstraction to allow the programming of switches to be independent of the underlying hardware. While this goals are very different to those of this project, they lead to the development of a programming language that has many desirable features.

A P4 program is comprised of five components:

- **Headers** The sequence and structure of packets
- **Parsers** Definition of how headers should be identified and validated
- **Tables** Packet matching tables for packet processing
- **Actions** Triggered by matching a table entry
- **Control Programs** Defines ordering and flow between matching (and action) entries

Of these, header definitions are of interest. These are specified by declaring an ordered list of field names, together with their bit widths. For example, the Ethernet header is specified as:

```
header ethernet {
  fields {
    dst_addr : 48; // width in bits
```

```
    src_addr : 48;
    ethertype : 16;
  }
}
```

[Note: the P4 paper doesn't focus on the header specification language; the P4 manual is a better resource. The language has also changed considerably since this paper]

Paper: "*OpenState: Programming Platform-independent Stateful OpenFlow Applications Inside the Switch*" – Bianchi et al. [6]

Review 23A from our EPIQ workshop submission suggested that this is relevant.

The software defined networking (SDN) architecture is comprised, conceptually, of smart, centralised controllers for forwarding behaviour dumb, commodity switches. In the event that complex, stateful forwarding actions need to be taken, packets are re-routed through controllers that can apply (relatively) complex logic. The motivation for OpenState is that there are scenarios where this architecture isn't sensible: sometimes, the forwarding behaviour only depends on local (i.e., at the switch level) state, and not on the global state that can be accessed by controllers. By allowing more logic to be performed at each switch, the costs associated with re-routing packets are removed, allowing packets to be processed at line-rate. The authors give *port knocking* as an example of the type of forwarding behaviour they would like to implement within a switch, rather than have handled at a central controller. Essentially, port knocking works by having a client send packets to a pre-determined sequence of ports, after which connections to a firewalled port will be permitted. This logic is relatively straightforward: each incoming packet changes the flow's state, either resetting it to the starting state, or moving it to the next state, if the incoming packet matched the specified port. At present, each packet would be rerouted to a controller for this logic to be applied: with the modifications proposed in OpenState, this would be performed at the switch.

The central contribution of the OpenState paper is an abstraction that allows for the stateful processing of flows to be described. This abstraction is based on a simplified eXtended Finite State Machine (XFSM) (a Mealy machine). A Mealy machine is comprised of a 4-tuple, (S, I, O, T), where S is a finite set of states, I is a finite set of events (input), O is a finite set of actions (output), and T is a transition function that maps $< state, event >$ pairs into $< state, action >$ pairs. The paper specifies how the Mealy machine abstraction can be implemented, using the abstractions that already exist in OpenFlow.

The authors conclude by explaining that the existing abstractions in OpenFlow are constraints on the abstraction they picked. In particular, they highlight that at present, all events are driven by the arrival of packets: there is no support for timers, or other events, that might change a flow's state. Finally, they highlight that the lack of persistent memory ("registers") means that it is not possible to support some applications. This is a restriction of the Mealy machine abstraction that they chose.

The essential relevance of this, and [5], would seem to be in the extensible finite state machine abstraction. A similar abstraction could also apply to managing parsing state or other contextual information between packets, and directing the operation of a protocol parser. Or, of course, in representing the state of a protocol handshake.

Paper: "*Open Packet Processor: a programmable architecture for wire speed platform-independent stateful in-network processing*" – Bianchi et al. [5]

Review 23A from our EPIQ workshop submission suggested that this is relevant.

Building on the OpenState paper, the authors (a superset of those of the OpenState paper) highlight applications (e.g., port scan detection) where a Mealy machine is an unsuitable abstraction, owing to its lack of memory registers. To support these applications, the authors describe the Open Packet Processor, which is based on a full eXtended Finite State Machine abstract. Essentially, where the Mealy machine abstraction is a 4-tuple (as above), the full abstraction is a 7-tuple: (I, O, S, D, F, U, T), where S, I, O, and T, remain the same, and where D is a set of registers, F is a set boolean predicates on registers, and U is a set of functions that update registers. The authors demonstrate the feasibility of implementing their abstraction, and give programming examples, showing how relatively complex applications can be implemented.

As with [6], this is building hardware independent in-switch packet processing abstractions to extend OpenFlow, using finite state machines and stored state tables as the underlying programming abstraction. A similar abatraction can likely be used as the to our work.

P4 looks to be a more general-purpose approach to the same problem, and has much broader traction in the community.

Paper: "*Automatic function test generation using the extended finite state machine model*" – Cheng et al. [14]

The underlying formalism used by [6] and [5]. The concept of using finite state machines to model the behaviour of a protocol is clearly important and relevant. Whether this specific formalism is appropriate is an open question.

Paper: "*The IVy language*" – Microsoft [27]

To do – Write up notes on IVy

See `https://mailarchive.ietf.org/arch/msg/quic/zMIMd_pEByo2oghhLbaJrNW-QOQ`.

Paper: "*Abstract Syntax Notation One (ASN.1): Specification of basic notation*" – ITU-T (Information technology) [32]

ASN.1 is a language for defining the structure of data. It includes a number of simple data types, and notation for referencing and specifying values for these. Importantly, this notation does not constrain the encoding format for the data being specified. This is handled by supplementary encoding rules: the "transfer syntax" – e.g., Basic Encoding Rules (BER), or Packed Encoding Rules (PER). The focus here is on the abstract notation, and the semantics it supports.

ASN.1 supports the general technique of defining a small core of simple types, and allowing these to be combined in various ways to define new types. These new types can be defined, for example, by composing an ordered sequence of the existing types, or using other structures or relationships. To allow for the unambiguous encoding of these types, they are assigned a tag: these are helpers for parsers of the language, rather than for human readers.

An instance of the ASN.1 notation (defining types, values, sets, etc) is a module. Modules definitions being with this syntax:

```
ModuleIdentifier DEFINITIONS ::= BEGIN
```

Modules contain type definitions:

```
ModuleIdentifier DEFINITIONS ::= BEGIN
    TypeIdentifier ::= SEQUENCE {
        label     INTEGER,
        label2    BOOLEAN
    }
END
```

ASN.1 allows for values to be constrained:

```
ModuleIdentifier DEFINITIONS ::= BEGIN
    TypeIdentifier ::= SEQUENCE {
        label     INTEGER(0..15),
        label2    BOOLEAN
    }
END
```

Additionally, definitions can be reused and extended:

```
ModuleIdentifier DEFINITIONS ::= BEGIN
    TypeIdentifier ::= SEQUENCE {
        label     INTEGER,
        label2    BOOLEAN
    }

    TypeIdentiferExtension ::= SEQUENCE {
        label3    SEQUENCE OF TypeIdentifier,
        ...
    }
END
```

The ".." is part of the language: it says that the TypeIdentifierExtension specification might contain additional fields in later versions.

Paper: "*A language-based approach to protocol construction*" – Basu et al. [3]

Basu et al. describe Promela++, an extension to the Promela protocol validation language. Promela++ adds constructs for specifying layered protocols and for composing these protocols into a stack. In addition, it provides support for the encapsulation of protocol state, and message headers. Again, the focus here is on the language itself, and the semantics it supports, rather than on the tooling that is described alongside it.

Promela++ is syntactically similar to C, but with a number of modifications and constraints. Importantly, Promela++ separates the lower-level network access and data handling components from the protocol-layer control structures. This has a number of advantages, allowing the low-level components to be hidden from the protocol-level validator.

In Promela++, a protocol layer is comprised of user-defined types that encapsulate the state of the protocol layer, and the message header used by the protocol layer. These are flagged using the `state` and `message` keywords respectively. Of interest here is the message header defintion language.

Message headers are expressed as:

```
message UAM_msg {
  UAM_short        am,
  ...
};
```

Paper: "*CSN. 1 Specification, Version 2.0*" – Mouly [28]

The Concrete Syntax Notation, CSN.1, is a language that was introduced by 3GPP, to aid in the description of the *encoded* format of layer 3 GSM messages. Previously, these messages were described with various ad hoc methods, including textual descriptions of the meaning of each bit. The goals of CSN.1, then, are to unify the way in which these messages are described, using a formal language, and in so doing, encourage the development of tooling that could generate encoding/decoding programs.

CSN.1 operates over a logical bit stream, of arbitrary length. A CSN.1 definition describes the transmission syntax, defining which sequences of bits are valid. In addition, components of the bit stream are labelled. However, CSN.1 does not allow for any relationships between these different components to be modelled. The meaning of each value is not defined; no attempt is made to constrain a given message to carry a set of values that all make sense together.

As an example:

```
< TEST > ::= 00001111;
```

Here, `TEST` has a fixed value (15).

```
< TEST_OR > ::= 0000 {1111 | 0000};
```

In this example, `TEST_OR` can take on two possible (8-bit) values: 0 or 15. However, there is no notation to define when `TEST_OR` should take on either of those values: that is entirely application dependent, and cannot be modelled here.

Paper: "*Concise data definition language (CDDL): a notational convention to express CBOR and JSON data structures*" – Birkholz [7]

To do – write up discussion of draft-ietf-cbor-cddl-06

Paper: "*Concise Binary Object Representation (CBOR)*" – Bormann and Hoffman [10]

The Concise Binary Object Representation (CBOR) has the following design goals:

- It must be able to unambiguously encode most of the data formats used in Internet standards;
- CBOR encoders/decoders must be supported by resource-constrained hosts;
- Data must be independently decodable (i.e., without a schema);
- The serialisation format must be reasonably compact;
- The format must be converted to and from JSON;
- The format must be extensible, with the extended representation decodable by earlier decoders.

The focus on supporting encoding/decoding by resource-constrained hosts, the compactness of the format, and on extensibility with backwards compatibility with existing decoders make it distinct from other formats, such as ASN.1.

In CBOR, encodings are split into *data items*, where the initial byte of each data item specifies its type: unsigned integer, negative integer, byte string, unicode string, array, map, tag, or floating-point number. Most of these types adopt their conventional definition.

Tags are optional data items that precede another data item, giving it additional semantic meaning. Decoders do not need to interpret tags, and so their primary purpose is to define common data types, and provide optional hints to generic CBOR decoders.

The CBOR specification provides a mapping to and from JSON.

TO support extensibility, CBOR provides three codepoint spaces that are underutilised, and so can be used in future revisions.

Paper: "*MessagePack*" – Sadayuki Furuhashi [19]

To do – write up discussion of MessagePack

Paper: "*Protocol Buffers*" – Google [21]

To do – write up discussion of protocol buffers

1.2 Packet parser design principles

Paper: "*Design principles for packet parsers*" – Gibb et al. [20]

In this paper, the authors discuss the design tradeoffs present in packet parsers, and identify the design principles that exist. Additionally, they describe a parser generator. As part of this parser generator, a language to describe packet formats is defined. As an example, the IPv4 header is described as:

```
ipv4 {
  fields {
    version        :  4,
    ihl            :  4,
    diffserv       :  8 : extract,
    totalLen       : 16,
    identification : 16,
    flags          :  3 : extract,
    fragOffset     : 13,
    ttl            :  8 : extract,
    protocol       :  8 : extract,
    hdrChecksum    : 16,
    srcAddr        : 32 : extract,
    dstAddr        : 32 : extract,
    options        : *,
  }
  next_header = map(fragOffset, protocol) {
     1 : icmp,
     6 : tcp,
    17 : udp,
  }
  length = ihl * 4 * 8
  max_length = 256
}
```

This description includes the name and size of each field, a label for fields that should be extracted, a mapping between field values, and for variable-length headers, a definition of length derived from field values.

To do – Expand

Paper: "*Writing parsers like it is 2017*" – Chifflier and Couprie [15]

In this paper, the authors present a Rust-based combinator approach to parser design. This is motivated by the prevalence of memory corruption bugs in parsers implemented in low-level languages. Many of the issues raised by the authors result from the use of unsafe languages, such as C, for parser implementation, and they recommend changing to a safe language, such as Rust, to solve many of these.

To do – Expand

1.3 Network protocol DSLs & motivation

Paper: "*Domain specific languages (DSLs) for network protocols*" – Bhatti et al. [4]

To do – Write up notes

Paper: "*The Bugs We Have to Kill*" – Bratus et al. [12]

To do – Read and write up

Paper: "*Noise in specifications hurts.*" – Bormann [9]

The paper does a translation of certain specifications from JSON to CDDL.
It shows clearly that syntactic complexity is a problem is specifications. Ease of reading is important.

Paper: "*JSON Data Definition Format (JDDF)*" – Carion [13]

The paper defines a JSON-based format for describing data formats.

Semantics are sensible, and generally map readily onto the format we're describing.

A good example of why syntax matters for protocol definition languages [9]. The result makes sense, but is ugly and not very human readable.

Paper: "*YANG object universal parsing interface*" – Petrov [29]

Another format for describing data formats, this time using YANG.

Paper: "*Thrift: Scalable Cross-Language Services Implementation*" – Slee at al. [33]

Interface and service definition language, originally from Facebook, now open source via the Apache Foundation.

Defines a type system for describing interfaces. - Obvious primitive types. - Structs - List, Set, and Map - Services and Exceptions

The set of data formats this can represent looks more limited than those we consider.

Good story about versioning.

This is solving a related, but different, problem to us. It's probably best thought of as a simplified version of CORBA or COM.

1.4 Summary

Language/Feature	Primitive types	Composite types	Value constraints	Inheritance	Derived properties
PacketTypes	•	•	•	•	•
Melange	•	•	•	•	
PADS	•	•	•		
DataScript	•	•	•		
P4	•	•	•	•	•
ASN.1	•	•	•		
Promela++	•	•	•		
CSN.1	•	•			

2 LangSec

Language-theoretic security (LangSec)[1] is a design philosophy that is underpinned by the belief that software bugs are a result of languages and architectures that prevent formal verification (i.e., that are undecideable). LangSec aims to produce verifiable parser implementations, that are free from many classes of bugs and vulnerabilities associated with incorrect parsing.

Paper: "*Writing parsers like it is 2017*" – Chifflier and Couprie [15]

In this paper, the authors present a Rust-based combinator approach to parser design. This is motivated by the prevalence of memory corruption bugs in parsers implemented in low-level languages. Many of the issues raised by the authors result from the use of unsafe languages, such as C, for parser implementation, and they recommend changing to a safe language, such as Rust, to solve many of these.

To do – Expand

Paper: "*Caradoc: a pragmatic approach to PDF parsing and validation*" – Endignoux et al. [17]

Explores the structural properties (rather than specific features) of the PDF format that allow for its use as a malware carrier. The authors propose a subset of the format's syntax that avoids common errors. The define a formal language for this syntax, and show that data consistency can be validated using a type checker.

The authors choose to propose an LR(1) grammar, restricting the syntax to those structures that can be parsed linearly (i.e., with no backtracking). There are a number of issues with this (e.g., implications for the elements that the restricted format can contain), but the broader takeaway is that some provable property of the grammar is prioritised ahead of flexibility. We want to do the same, albeit with a different set of tradeoffs/choices.

The approach of identifying particularly problematic components of the PDF structure, and excising those from the restricted format, is reasonable.

[1] `http://langsec.org/bof-handout.pdf`

Paper: "*Grammatical inference and language frameworks for LANGSEC*" – Wood and Harang [34]

To do – Write up

Paper: "*LangSec revisited: input security flaws of the second kind*" – Poll [30]

To do – Write up

Paper: "*Mind your language (s): A discussion about languages and security*" – Jaeger and Levillain [22]

This paper discusses the idea that the mechanisms and constructs that a programming language contains can provide intrinsic security properties to the programs that are written, and the tools that support their development. The paper discusses the pros and cons of a number of programming paradihms, methods, and mechanisms, to understand their impact on security.

This discussion is relevant to the project: it is worth considering the instrinsic properties that arise from the methods by which protocols are specified. With guidance (or enforcement) from specification languages or tooling, the set of protocols that can be specified could be narrowed to those that require parsers that have desirable instrinsic properties. This requires thinking about the properties of the parser: if, for example, linear parsing is desirable, this enforces a restriction on the possible packet layouts. We should consider whether such restrictions are, overall, beneficial.

Paper: "*Nail: A practical interface generator for data formats*" – Bangert and Zeldovich [2]

Writing input processing code is difficult. One approach is to specify a grammar for the input data format, and then use a parser generator to help with the generation of input processing code. In motivating Nail, the authors outline three broad issues with this approach. First, parser generators generally produce an abstract syntax tree; this data structure is not the same as the internal representation held by the application, requiring some code (*semantic actions*) to be written; writing this code is as challenging as other input processing code. Second, applications are likely to need to produce output in the same format as the input: most parser generators focus only on input processing. Finally, input data formats (and especially binary formats) can have *structural dependencies* (i.e., fields that are dependent on others) that are difficult to express. Further, they lead to programmers having to write control code (e.g., to move the parser's input stream to allow for dependent fields to be parsed in the correct order).

To address these challenges, the authors describe Nail, a parser generator that reduces how much additional code needs to be written when using a grammar-based parser. Nail does so by (i) reducing the expressiveness of its grammar language; Nail's representation, rather than an AST, is a data structure that the application can use directly; (ii) using the same internal representation to invert the parser, and generate valid output; and (iii) providing support for structural dependencies.

A prototype, written in C, is provided, alongside an example for parsing and generating DNS packets.

Paper: "*Nom, a byte oriented, streaming, zero copy, parser combinators library in rust*" – Couprie [16]

To do – Write up

Paper: "*Parsifal: A Pragmatic Solution to the Binary Parsing Problems*" – Levillain [23]

Parsifal is a framework for describing parsers in OCaml. It consists of PTypes, which are comprised of an OCaml type, a parser function (that takes a binary representation of an object, and transforms it into the OCaml type), a serialisation function (that does the inverse of the parser function) and a representation function (that produces a representation of the OCaml type). There are three kinds of PType: basic types, provided by the standard library, keyword-assisted PTypes, like structures, that are described in a DSL, and custom PTypes.

The basic types are broadly similar to those that we've identified (e.g., enums, structs, choices/unions, and aliases. In addition, there are basic types that support ASN.1 DER components.

Parsifal also includes PContainers. The container abstraction contains a PType, and makes it possible to automate some process that has to be done at parsing or serialisation time (so, for example, encoding or compression). The example given is a struct with a field that is a compressed string. The compressed string is in a PContainer which captures the compression process. This is similar to our IR's use of initialisation and helper functions, and so is worth further thought.

The author concludes with a set of lessons learned. This includes several properties of packet formats that are problematic for the generation of parsers. For example, non-linear parsing is highlighted as being problematic.

Paper: "*Protocol state machines and session languages: specification, implementation, and security flaws*" – Poll et al. [31]

To do – Write up

Paper: "*Taming the Length Field in Binary Data: Calc-Regular Languages*" – Lucks et al. [24]

To do – Write up

Paper: "*The Pitfalls of Protocol Design: Attempting to Write a Formally Verified PDF Parser*" – Bogk and Schöpl [8]

Motivated by the difficulty of writing correct input processing code, the authors attempt to write a formally-verified PDF parser using Coq. They found this to be an "extremely hard" exercise, entirely as a result of the PDF specification, which specifies a language that is not context free. The main conclusion is that the definitions of protocols, file formats and other input languages must be complete, unambiguous, and parseable. Further, the language should be context-free to make sure that formal analysis is possible, and to improve the reliability of implementations.

This last constraint – being context-free - is an interesting one, with implications on the possible binary packet formats that are allowed.

3 Parsing IETF protocols

In this section, we look at the features required of a packet format definition language, when describing the syntax of various IETF protocols. Such a language must allow for a parser to be generated based upon the definition: enough information should be available to validate the syntax of a given packet, against the definition. To do this, we use the concept of *contexts*. The parsing process makes use of two contexts: a per-packet context, and a per-flow context. Contexts contain information from other, earlier fields (in the case of the per-packet context) or from other, earlier packets (in the case of the per-flow context) *that is necessary to parse other fields*. The per-packet context contains the values of earlier fields, and these can be used to constrain the value or properties of later fields. For example, an IP header has a `version` field; the value of this field determines the layout of the packet. The per-flow context can be added to by the parser, acting on earlier packets, or by an out-of-band process. These values can be used to constrain the value or properties of fields in later packets. The per-flow context is particularly useful for cryptographic protocols, where a key is provided as part of a message exchange earlier in the flow (or out-of-band).

It is important to note here that it is easy to see that these contexts could be used to enforce semantic properties of packets, rather than the syntactic validation that parsing requires. For example, the per-flow context might be used to determine that packets out-of-sequence are invalid. However, this is out-of-scope: constraints on packet values, based on data from contexts, should only be used for parsing – that is, to determine if a packet's format is valid, rather than determining if the contexts of each field are valid.

3.1 IP

The IP header is comprised of a 4 bit `version` field. The rest of the header's format is determined by the value of this field.

IPv4

An IPv4 header has the following fields:

- `IHL`: 4 bits
- `DiffServ`: 6 bits
- `ECN`: 2 bits
- `Total Length`: 16 bits
- `Identification`: 16 bits
- `Flags`: 3 bits
- `Offset`: 13 bits
- `TTL`: 8 bits
- `Protocol`: 8 bits
- `Checksum`: 16 bits
- `Source Addr`: 32 bits
- `Dest Addr`: 32 bits
- `Options`: variable length (incorporates padding to 32 bit boundary)
- `Payload`: variable length

Constraints:

- The length of the Options field depends on the IHL field: `len(Options) = IHL*4 - 20` (where the length is in bytes).

- The Options field contains zero or more variable length Options, with their own sub-format:
  - `Copy` 1 bit
  - `Class` 2 bits
  - `Number` 5 bits
  - `Length` 8 bits
  - `Data` variable length

  The number of Options is constrained both by the length of the field, and by the length of each Option. The Options field needs to pad to a 32 bit boundary. Further, the header can be at most 60 bytes long, leaving a maximum of 40 bytes for the Options field.
- The value of the Protocol field determines how the Payload field should be parsed.

Notes:

- The DiffServ and ECN fields were originally specified as a single 8 bit Type of Service (ToS) field. How should a definition language support changes in specification, where those changes happen between different documents?
- Parsing the Options field is interesting: it contains a zero or more variable length Options. The definition language should support sequences of this type, where the length of the sequence can only be determined by parsing each element in turn.

3.1.1 IPv6

An IPv6 header has the following fields:

- `Traffic Class`: 8 bits
- `Flow Label`: 20 bits
- `Payload Length`: 16 bits
- `Next Header`: 8 bits
- `Hop Limit`: 8 bits
- `Source Addr`: 128 bits
- `Dest Addr`: 128 bits
- `Payload`: variable length

Constraints:

- The value of the Next Header field determines how the Payload field should be parsed.

Notes:

- IPv6 headers don't have an Options field. Extension headers play much the same role, but these are handled in a much cleaner way: they are chained onto the IPv6 header using the Next Header field (with the chain ending with a upper-layer protocol, or a No Next Header value). While extensions to the IPv6 protocol, for parsing, it makes sense to treat them as a separate upper-layer protocol.

3.2 UDP

A UDP header has the following fields:

- `Source Port`: 16 bits
- `Dest Port`: 16 bits
- `Length`: 16 bits
- `Checksum`: 16 bits
- `Payload`: variable length

Notes:

- The UDP header is straightforward; no constraints based on other fields or earlier packets.

3.3 TCP

A TCP header has the following fields:

- `Source Port`: 16 bits
- `Dest Port`: 16 bits
- `Sequence Number`: 32 bits
- `ACK Number`: 32 bits
- `Data Offset`: 4 bits
- `Reserved`: 3 bits
- `Flags`: 9 bits
- `Window Size`: 16 bits
- `Checksum`: 16 bits
- `URG Pointer`: 16 bits
- `Options`: variable length (incorporates padding to 32 bit boundary)
- `Payload`: variable length

Constraints:

- The length of the Options field depends on the Data Offset field: `len(Options) = Data Offset - 5` (where the length is in 32-bit words).
- The Options field contains zero or more variable length Options, with their own sub-format:
  - `Option-Kind` 8 bits
  - `Option-Length` 8 bits
  - `Option-Data` variable length

  The number of Options is constrained both by the length of the field, and by the length of each Option. The Options field needs to pad to a 32 bit boundary. Further, the header can be at most 60 bytes; leaving a maximum of 40 bytes for the Options field.

3.4 DCCP

A DCCP header has the following fields:

- `Source Port`: 16 bits
- `Dest Port`: 16 bits
- `Data Offset`: 8 bits
- `CCVal`: 4 bits
- `CsCov`: 4 bits
- `Checksum`: 16 bits
- `Res`: 3 bits
- `Type`: 4 bits
- `eXtended Sequence Number Flag`: 1 bit
- `Reserved`: 0 (X=1) or 8 (X=0) bits
- `Sequence Number`: 48 (X=1) or 24 (X=0) bits
- `Payload`: variable length

Constraints:

- The layout of the packet after the X field depends on its value; if X=1, then there is an 8 bit Reserved field, followed by an extended, 48 bit, sequence number. If X=0, then there is no Reserved field, and the sequence number is 24 bits.

3.5 SCTP

An SCTP header has the following fields:

- `Source Port`: 16 bits
- `Dest Port`: 16 bits
- `Verification Tag`: 32 bits
- `Checksum`: 32 bits
- `Payload`: variable number of variable-length chunks

Notes:

- The payload is in the form of chunks, which have the following format:
  - `Chunk Type`: 8 bits
  - `Chunk Flags`: 8 bits
  - `Chunk Length`: 16 bits
  - `Chunk Data`: variable length (padded to 32 bit boundary)
- Chunk formats are standardised; again, any definition language used would need to support this type of extensibility.
- There are a variable number of chunks, of variable length, so the number of chunks is not known before they are parsed.

3.6 QUIC

QUIC has two header types: long (for packets sent before version negotiation) and short (after version negotiation). The most significant bit in the first octet determines whether the header is long (=1) or short (=0).

The long header has the following format:

- `Header Form`: 1 bit (set to 0)
- `Long Packet Type`: 7 bits
- `Version`: 32 bits
- `Destination Connection ID Length`: 4 bits
- `Source Connection ID Length`: 4 bits
- `Destination Connection ID`: 0 or 32 to 144 bits
- `Source Connection ID`: 0 or 32 to 144 bits
- `Payload Length`: variable length integer
- `Packet Number`: 32 bits
- `Payload`: variable length

Constraints:

- The Destination and Source Connection ID fields have their length defined by the Destination and Connection ID Length fields. The length fields are encoded as 4-bit unsigned integers; if they are zero, then the connection ID fields are zero. However, if they are non-zero, then the related connection ID field this value plus 3. As a result, the length is either 0, or between 32 to 144 bits. The packet header specification language needs to be able to express this logic.
- The payload length field is a variable length integer. This field is either 1, 2, 4, or 8 octets long, depending on the value of the two most significant bits of the first octet (they encode the base 2 logarithm of the length of the field). The remaining bits are used to encode the value.

The short header has the following format:

- `Header Form`: 1 bit (set to 0)
- `Key Phase Bit`: 1 bit

- `Third Bit`: 1 bit (set to 1)
- `Fourth Bit`: 1 bit (set to 1)
- `Google QUIC Demultiplexing Bit`: 1 bit (set to 0)
- `Reserved`: 1 bit
- `Short Packet Type`: 2 bits
- `Destination Connection ID`: 0 to 144 bits
- `Packet Number`: 8, 16, or 32 bits
- `Protected Payload`: variable length

Constraints:

- The Destination Connection ID is the Source Connection ID received by the endpoint. Given that this is a variable length field, it can only be parsed if we have the Source Connection ID Length field received earlier in the flow (as part of the long header).
- The Packet Number field is either 8, 16, or 32 bits, depending on the value of the short packet type field.
- The Protected Payload field is protected using authenticated encryption. The key is derived from the TLS handshake. The nonce, however, is formed from the packet protection invariant, and the packet number. The packet number carried in the packet, though, carries only the least significant bits of the full packet number needed for determining the cryptographic nonce. Therefore, the parser must keep track of the highest sequence number successfully authenticated.

Version Negotiation packet

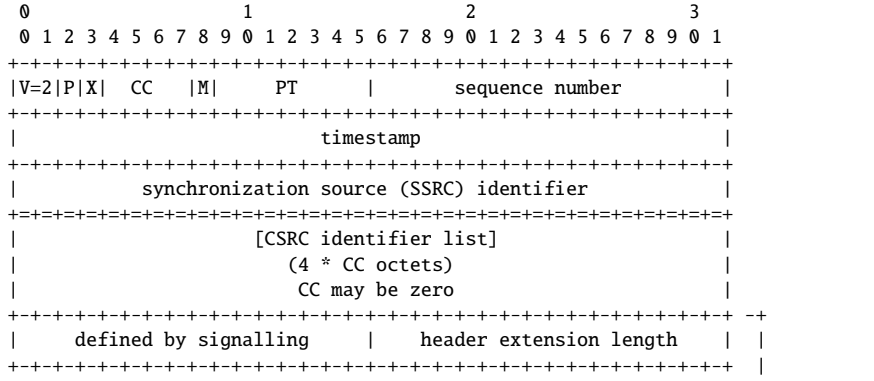
The version negotiation packet format is:

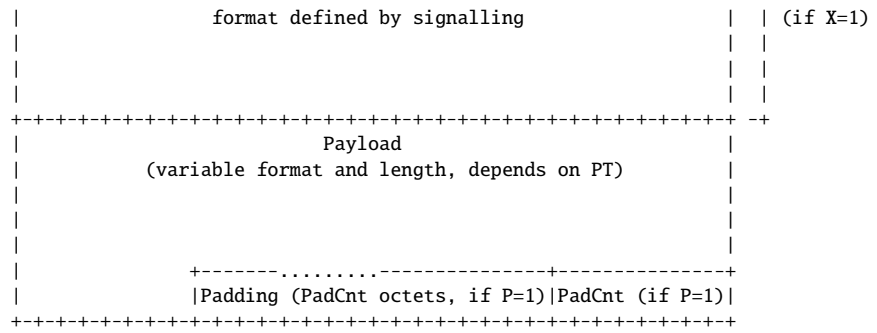
- `Header Form`: 1 bit (set to 1)
- `Unused`: 7 bits
- `Version`: 32 bits
- `Destination Connection ID Length`: 4 bits
- `Source Connection ID Length`: 4 bits
- `Destination Connection ID`: 0 or 32 to 144 bits
- `Source Connection ID`: 0 or 32 to 144 bits
- `Supported Versions`: 1 or more 32 bit fields

Notes:

- This is not the long header format, but it does have the same value for the first bit. Therefore, this bit alone is not sufficient to determine how the packet should be parsed: the Version field must be set to 0.

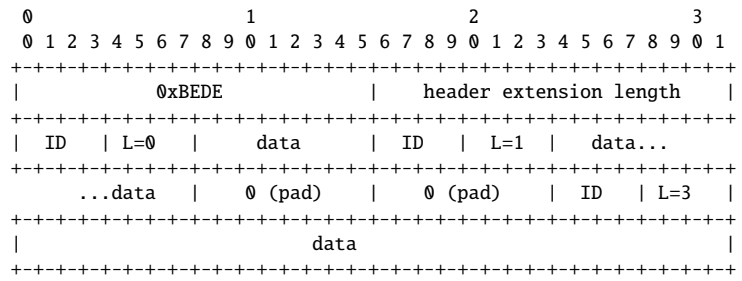
3.7 RTP





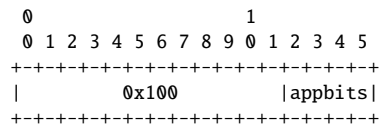
- Some fields are optional, and signalled by the value of earlier header fields
- Format of some fields depends on the value of earlier header fields, with the meaning of those earlier fields being signalled out-of-band
- Presence and length of some fields depends on later fields

Format of the header extension needs to be signalled in SDP. The only standard format is RFC5285:



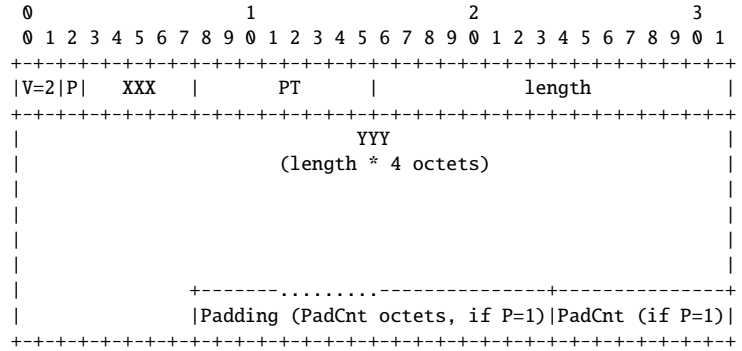
- 0xBEDE ("the first version of this specification was written on the feast day of the Venerable Bede").
- ID values define the type of extension, meaning signalled out of band
- L is length of following data
- L+1 data bytes follow
- ID=0, L=0 is a single byte of padding and is ignored
- MUST pad to a 32 bit boundary

There's also a variant where ID and L are one octet each, and the first 16 bits of the header are:



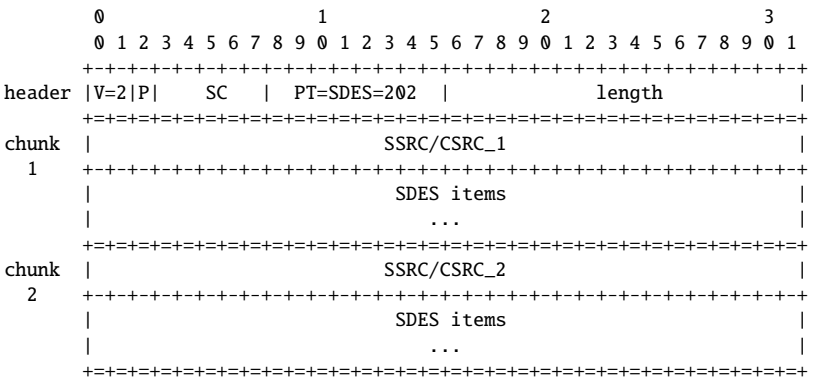
where `appbits` are 4 bits that "are application dependent and MAY be defined to be any value or meaning; this topic is outside the scope of this specification".

3.8 RTCP

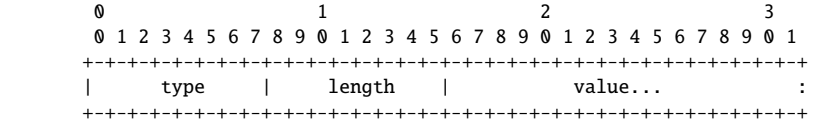


- Meaning of XXX and YYY depends on PT. This implies that the meaning of a field depends on the value of a later field
- RTCP packets are often sent as a *compound packet* where multiple RTCP packets are stacked into a single UDP packet. There are several rules about how this is done:
  - The first packet in the compound packet MUST be an SR or RR. If there are additional RR packets, they SHOULD follow the initial SR/RR packet.
  - The compound packet MUST include an SDES packet, and that SDES packet MUST include a CNAME item.
  - If padding is added, it MUST only be added to the last RTCP packet in the compound packet. That is, P=0 on all but the last packet.

Of the packet types defined in RFC 3550, only the SDES is complex:



where SC is the count of the number of chunks. Each chunk comprises a list of items, where the format of an item is:



where `length` is the length of the `value` in octets (may be zero). The list of items in each chunk MUST be terminated by one or more null octets, the first of which is interpreted as an item type of zero to denote the end of the list. No length octet follows the null item type octet, but additional null octets MUST be included if needed to pad until the next 32-bit boundary. Note that this padding is separate from that indicated by the P bit in the RTCP header. A chunk with zero items (four null octets) is valid but useless.

RTCP XR has many sub-packet formats. There doesn't look to be anything unusual in the formatting beyond standard RTCP packets, however.

3.9 RTSP

3.10 NTP

The NTP (v4) packet format is:

- `Leap Indicator`: 2 bits
- `Version Number`: 3 bits
- `Mode`: 3 bits
- `Stratum`: 8 bits
- `Poll`: 8 bits
- `Precision`: 8 bits
- `Root Delay`: 32 bits (NTP short format)
- `Root Dispersion`: 32 bits (NTP short format)
- `Reference Timestamp`: 64 bits (NTP timestamp format)
- `Origin Timestamp`: 64 bits (NTP timestamp format)
- `Receive Timestamp`: 64 bits (NTP timestamp format)

- `Transmit Timestamp`: 64 bits (NTP timestamp format)
- `Extension Fields`: variable length; format below
- `Key Identifier`: 32 bits
- `Message Digest`: 128 bits

Notes:

- The packet may contain one or more extension fields, with the following format:
  - `Field Type`: 16 bits
  - `Length`: 16 bits (length of entire extension field in octets)
  - `Value`: variable length
  - `Padding`: variable length as required to pad to 32 bit word
- An interesting component here is the variable number of variable length extension fields. More specifically, the padding field is included in the length – this makes parsing more challenging.

3.11 HTTP/2

HTTP/2 uses frames, with the following format:

- `Length`: 24 bits
- `Type`: 8 bits
- `Flags`: 8 bits
- `Reserved`: 1 bit
- `Stream Identifier`: 31 bits
- `Payload`: variable length

4 Proposed packet format description language

4.1 What constraints need to be modelled carefully?

Variable length fields
: Variable length fields at the end of packets are trivial: parsing continues until no data remains. However, variable length fields anywhere else in the packet require information about the length of the packet, or from other fields. For example, in the IP header, the length of the Options field is determined by the value of the IHL field. Any description language needs to allow for the width of fields to be determined by other fields (including, possibly, later fields in the packet).

  Additionally, QUIC packets carry connection IDs. These are variable length fields, but their length is not necessarily in the same packet (i.e., it is in the long packet format, but not in the short packet format).

Conditional structuring
: A number of the protocols have version fields (e.g., IP) or header format bits (e.g., QUIC) that define the structure of the packet after that bit.

Encrypted fields
: In QUIC, the frames contained within each packet cannot be parsed without first being decrypted. This represents a shift from how protocols have typically used cryptography, where application data was encrypted, and therefore could be passed to the application as a "blob", without further decryption: HTTPS works in this way – headers are sent in the clear, while the body is encrypted. The encryption of other header fields means that parsers are incomplete if they do not support cryptography primitives.

Out-of-band data
: Some fields cannot be parsed without data that has been exchange out-of-band (i.e., not within the flow that the parser's scope). For example, RTP packets contain optional header extension fields, but the format of these is signalled by SDP, not RTP: we require the out-of-band data from the SDP exchange to fully parse the RTP packet.

Computed fields
: QUIC includes a Packet Number field, which is required to decrypt the contents of the Protected Payload field. However, this field only contains the least significant bits of the packet number; as a result, more information is required to compute the actual packet number, for decrypting the payload field.

4.2 What concepts do we need to introduce?

Contexts

From the above, it is clear that it is common for the parser to values from fields that have been parsed earlier in a flow (e.g., from a handshake packet as part of the same connection), or from out-of-band exchanges, as is the case for header extensions in RTP. A solution to this would be *contexts*: essentially, a key-value store that the description language can model additions to and retrievals from.

To do – How should out-of-band additions be described? To do – How should failed retrievals be handled?

Interfaces

In QUIC, there are two fields that require some computation in order to fully parse each packet: packet numbers and protected payloads. It is beyond the scope of a format description language to model these computations, but we need some way of indicating that a computation needs to have been performed before parsing can proceed.

A solution to this would be for a set of function signatures to be provided as part of the definition.

To do – How should failed computations be handled? To do – Is there value in constraining the set of functions that can be defined? e.g., decrypt() – packet numbers suggests something more general is needed

4.3 Describing QUIC

```
packet_type := bit[7];
version := bit[32];
cid_len := bit[4];
var_enc := {
  bit[2]  length;
  bit[]   value;
} where {
  value#width = (2^length#value * 8) - 2;
}

full_packet_num := bit[62]
packet_num :=
    '0'  followed by bit[7]  packet_number
  | '00' followed by bit[14] packet_number
  | '01' followed by bit[30] packet_number;

decrypt :: (cryptobits[] enc_payload, full_packet_num pn) -> FRAME[]

frame_type := bit[8];

QUIC_PDU :=
    LONG_HDR
  | SHORT_HDR
  | VERSION_NEGOTIATION;

LONG_HDR := {
  bit          header_type;
  packet_type  type;
  version      ver;
  cid_len      dcid_len;
  cid_len      scid_len;
  bit[]        dcid;
  bit[]        scid;
  var_enc      payload_length;
  packet_num   packet_number;
  bit[]        payload;
} where {
  header_type#value = 1;
  dcid#width = dcid_len#value == 0 ? 0 : (dcid_len#value+3) * 8;
  scid#width = scid_len#value == 0 ? 0 : (scid_len#value+3) * 8;
  payload#width = 2^payload_length#value;
```

```
} onparse {
  context.scid_len = scid_len;
}

SHORT_HDR := {
  bit         header_type;
  bit         key_phase;
  bit         third_bit;
  bit         forth_bit;
  bit         google_demux;
  bit[3]      reserved;
  bit[]       dcid;
  packet_num  packet_number;
  cryptobit[] protected_payload; -> frame[] payload;
} where {
  header_type#value = 0;
  third_bit#value = 1;
  forth_bit#value = 1;
  google_demux#value = 0;
  dcid#length = context.scid_len#value == 0 ? 0 : (context.scid_len#value+3) * 8;
  payload = decrypt(protected_payload, packet_number);
}

VERSION_NEGOTIATION := {
  bit          header_type;
  bit[7]       unused;
  version      ver;
  cid_len      dcid_len;
  cid_len      scid_len;
  bit[]        dcid;
  bit[]        scid;
  version[]    supported_versions;
} where {
  header_type#value = 1;
  ver#value = 0;
  dcid#width = dcid_len#value == 0 ? 0 : (dcid_len#value+3) * 8;
  scid#width = scid_len#value == 0 ? 0 : (scid_len#value+3) * 8;
}

FRAME := {
    PADDING_FRAME
  | RST_STREAM_FRAME
  | CONNECTION_CLOSE_FRAME
  | ..};

PADDING_FRAME := {
  frame_type type;
} where {
  type#value = 0;
}
```

Notes

- This doesn't capture all (or even most) of QUIC's encryption/authentication behaviour, but contains enough to show most of the syntax and semantics of a description language.
- A basic principle is that types ultimately resolve to `bits`, either through type definitions, or method signatures.
- Type definitions are either: (i) variants (enums), e.g., `QUIC_PDU`, (ii) composite data types (structs), e.g., `LONG_HDR`; or (iii) simple data types (typedefs), e.g., `version`.

- For (ii), they aren't just composite data types; they are dependent types – the "where" block encodes values. This is inevitable, if variants are to be supported as shown: variants are disambiguated on the value of one or more of the fields.
- Simple data types are essentially typedefs at the moment, but it is easy to see how they could develop into something more like a Java class, where the set of operations that can be performed on a type are well defined.

References

[1] G. Back. Datascript-a specification and scripting language for binary data. In *International Conference on Generative Programming and Component Engineering*, pages 66–77. Springer, 2002.

[2] J. Bangert and N. Zeldovich. Nail: A practical interface generator for data formats. In *2014 IEEE Security and Privacy Workshops*, pages 158–166. IEEE, 2014.

[3] A. Basu, M. Hayden, G. Morrisett, and T. Von Eicken. A language-based approach to protocol construction. In *Proceedings of the Workshop on Domain Specific Languages*, pages 87–99, Paris, France, January 1997. ACM.

[4] S. Bhatti, E. Brady, K. Hammond, and J. McKinna. Domain specific languages (dsls) for network protocols. In *International Workshop on Next Generation Network Architecture (NGNA 2009)*, volume 165, page 166, 2009.

[5] G. Bianchi, M. Bonola, S. Pontarelli, D. Sanvito, A. Capone, and C. Cascone. Open packet processor: a programmable architecture for wire speed platform-independent stateful in-network processing. arXiv:1605.01977, May 2016.

[6] G. Bianchi, A. Caponi, M. Bonola, and C. Cascone. OpenState: Programming platform-independent stateful OpenFlow applications inside the switch. *Computer Communication Review*, 44(2):45–51, Apr. 2014.

[7] H. Birkholz, C. Vigano, and C. Bormann. Concise data definition language (CDDL): a notational convention to express CBOR and JSON data structures. Internet Engineering Task Force, November 2018. draft-ietf-cbor-cddl-06.

[8] A. Bogk and M. Schöpl. The pitfalls of protocol design: Attempting to write a formally verified pdf parser. In *2014 IEEE Security and Privacy Workshops*, pages 198–203. IEEE, 2014.

[9] C. Bormann. Noise in specifications hurts. In *Proceedings of the Internet of Things Semantic Interoperability Workshop*, San Jose, CA, USA, Mar. 2016. Internet Architecture Board.

[10] C. Bormann and P. Hoffman. Concise Binary Object Representation (CBOR). RFC 7049, Oct. 2013.

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[12] S. Bratus, M. L. Patterson, and A. Shubina. The bugs we have to kill. *; login:: the magazine of USENIX & SAGE*, 40(4):4–10, 2015.

[13] U. Carion. JSON data definition format (JDDF). Internet Engineering Task Force, Oct. 2019. Work in progress (draft-ucarion-jddf-03).

[14] K.-T. Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Design Automation, 1993. 30th Conference on*, pages 86–91. IEEE, 1993.

[15] P. Chifflier and G. Couprie. Writing parsers like it is 2017. In *2017 IEEE Security and Privacy Workshops (SPW)*, pages 80–92. IEEE, 2017.

[16] G. Couprie. Nom, a byte oriented, streaming, zero copy, parser combinators library in rust. In *2015 IEEE Security and Privacy Workshops*, pages 142–148. IEEE, 2015.

[17] G. Endignoux, O. Levillain, and J.-Y. Migeon. Caradoc: a pragmatic approach to pdf parsing and validation. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 126–139. Ieee, 2016.

[18] K. Fisher and R. Gruber. PADS: a domain-specific language for processing ad-hoc data. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 295–304, Chicago, IL, USA, June 2005. ACM.

[19] S. Furuhashi. Messagepack. Document available online, December 2018. `https://msgpack.org`.

[20] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design principles for packet parsers. In *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 13–24. IEEE Press, 2013.

[21] Google. Protocol buffers. Document available online, November 2018. https://developers.google.com/protocol-buffers/.

[22] E. Jaeger and O. Levillain. Mind your language (s): A discussion about languages and security. In *2014 IEEE Security and Privacy Workshops*, pages 140–151. IEEE, 2014.

[23] O. Levillain. Parsifal: A pragmatic solution to the binary parsing problems. In *2014 IEEE Security and Privacy Workshops*, pages 191–197. IEEE, 2014.

[24] S. Lucks, N. M. Grosch, and J. König. Taming the length field in binary data: Calc-regular languages. In *2017 IEEE Security and Privacy Workshops (SPW)*, pages 66–79. IEEE, 2017.

[25] A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan. Melange: Creating a "functional" Internet. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Lisbon, Portgual, Mar. 2007. ACM.

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

[27] Microsoft. The ivy language. Document available online, November 2018. http://microsoft.github.io/ivy/language.html.

[28] M. Mouly. *CSN. 1 Specification, Version 2.0*. Cell & Sys, 1998.

[29] I. Petrov. YANG object universal parsing interface. Internet Engineering Task Force, Nov. 2019. Work in progress (draft-petrov-t2trg-youpi-01).

[30] E. Poll. Langsec revisited: input security flaws of the second kind. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 329–334. IEEE, 2018.

[31] E. Poll, J. De Ruiter, and A. Schubert. Protocol state machines and session languages: specification, implementation, and security flaws. In *2015 IEEE Security and Privacy Workshops*, pages 125–133. IEEE, 2015.

[32] I.-T. Recommendation X.680 (08/15). Abstract syntax notation one (asn.1): Specification of basic notation, 2015.

[33] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. Whitepaper, Apr. 2007.

[34] K. N. Wood and R. E. Harang. Grammatical inference and language frameworks for langsec. In *2015 IEEE Security and Privacy Workshops*, pages 88–98. IEEE, 2015.