RecordFlux: Formal Message Specification and Generation of Verifiable Binary Parsers

Tobias Reiher[1], Alexander Senier[1], Jeronimo Castrillon[2], and Thorsten Strufe[2]

[1] Componolit, Dresden, Germany
{reiher,senier}@componolit.com
[2] TU Dresden, Dresden, Germany
{jeronimo.castrillon,thorsten.strufe}@tu-dresden.de

Abstract. Various vulnerabilities have been found in message parsers of protocol implementations in the past. Even highly sensitive software components like TLS libraries are affected regularly. Resulting issues range from denial-of-service attacks to the extraction of sensitive information. The complexity of protocols and imprecise specifications in natural language are the core reasons for subtle bugs in implementations, which are hard to find. The lack of precise specifications impedes formal verification.
In this paper, we propose a model and a corresponding domain-specific language to formally specify message formats of existing real-world binary protocols. A unique feature of the model is the capability to define invariants, which specify relations and dependencies between message fields. Furthermore, the model allows defining the relation of messages between different protocol layers and thus ensures correct interpretation of payload data. We present a technique to derive verifiable parsers based on the model, generate efficient code for their implementation, and automatically prove the absence of runtime errors. Examples of parser specifications for Ethernet and TLS demonstrate the applicability of our approach.

1 Introduction

Security issues are common in parsers of communication protocol implementations, and new vulnerabilities are found every day. Vulnerabilities caused by incorrect parsing exist on all protocol layers: from physical and network layer protocols like Bluetooth (BlueBorne [21]) over session-layer protocols like TLS (Heartbleed [19]) to application-layer protocols like SMB (EternalBlue [20]). Communication protocols are an increasingly worthwhile attack target, as more and more devices of our everyday life are connected to the Internet. Their reliability is especially important in business-critical, mission-critical and safety-critical software. Software that suddenly stops working is a potential threat to human life, be it in case of patients with an artificial heart or drivers steering their vehicles and braking using x-by-wire. While the problem is quite obvious for highly interconnected cars, even medical devices have at least an interface for software updates, which represent an attack surface for potential compromise by targeted attacks. Therefore, appropriate methods are needed to prevent the introduction of critical errors in protocol implementations.

Message formats of existing real-world protocols are often complex, but rarely formally specified. The simple syntax that is commonly used only defines the basic structure of a message. Additional properties, conditions, and relations between fields are just described in English prose. Such descriptions are imprecise and can easily be misunderstood by developers, which leads to implementation bugs. Lack of formal specification also prevents automatic checks and verification of the implementations.

Manual implementation has yielded 'shotgun parsers' that mix parsing and processing of messages, in the past. The consequence have been various critical vulnerabilities [12]. We assert that generating the parsing code from a formal grammar yields more cleanly separated implementations.

A recurrent cause of vulnerabilities is the widespread use of unsafe programming languages, like C++. Rust and other memory-safe languages have been developed to avoid memory corruptions. Using these languages is a clear progress towards security, but it does not prevent all errors at runtime.

Runtime errors like integer overflows or divisions by zero must still be handled explicitly. Negligence of the matter can have devastating effects, as reported for instance in [16]. Formal verification is the only convincing approach towards this end. Data and control flow analyses can prove their absence, and proving specific properties of software components is the only way to guarantee that unexpected errors do not occur at runtime.

In summary it becomes clear that a suitable process for the secure implementation of message parsers is needed. Concluding from the observations above, we pose the following requirements. At its heart, we need a simple, readable, and expressive domain-specific language (DSL) for a data format specification that is suitable for messages of existing real-world protocols. It shall also cover all invariants of the message parts. It is crucial that the generated code has been verified to be free of runtime errors, to enable its use in security-critical and safety-critical applications. To facilitate application in a wide range of areas, the generated code has to meet the performance requirements and resource limitations, even of embedded systems.

In this paper we introduce a generic approach for the specification of message formats and a methodology for creating verifiable parsers. Our main contributions are:

- We propose a DSL and model for the formal specification of message formats of existing real-world binary protocols, which covers all properties and dependencies of message parts by using invariants.
- We introduce a methodology for the automatic generation of parsers, for which the absence of runtime errors can be shown.
- We show the applicability of our approach on TLS 1.3 and the TLS Heartbeat protocol.

The rest of the paper is organized as follows: Section 2 gives an overview of related work. Section 3 introduces the model for the specification of messages. The design and implementation of the RecordFlux toolset is described in Section 4. The applicability is shown in Section 5 for two case studies. Section 6 gives a conclusion and an outlook for the future.

2 Related Work

In this section we describe related work for interface generators and generic parsers.

Interface Generators Interface generators like ASN.1 [15], XDR [28], or Protocol Buffers [4] are used for the development of programs which communicate with each other using serialized structured data. Although they are used to describe the message formats in various protocols or applications, they are not compatible to each other and lack the generality to specify messages of already existing protocols. Today's commonly used communication protocols are quite complex. Such protocols have grown historically, and therefore contain ambiguous idioms, like overlapping message fields that need to be parsed before their existence is clear. Their representation hence is impossible with the given interface generators.

Generic Parsers Generic parsers differ in the way how message formats are specified and which properties the generated code achieves.

One class are parser generators with a declarative description of the message structure. PacketTypes [17] and DataScript [8] use a type-based language to describe the layout of data formats. Binpac [26] is a declarative language for analyzing network protocols. GAPA [11] has a BNF-based specification language which matches the syntax commonly used in RFCs. Kaitai Struct [3] is YAML-based language to specify binary data formats.

Another class are parser combinators. They combine several existing parsers that are represented by functions into a single, new parser. Representatives are Hammer [2], a parsing library for binary formats written in C, attoparsec [25], a parser combinator library for Haskell, and nom [13], a parser combinator written in Rust that leverages Rust's strong type system and memory safety. Parser combinators may contain ambiguities in the grammar which are not reported at compile-time. Consequently, parser combinators do not meet our requirements.

Parser generators in contrast can provide means to prevent ambiguities before generating code. Many parser generators, however, use unsafe programming languages like C or C++: Binpac [26], PADS [14], Nail [10]. Even if the automatic nature of code generation alleviates some risk, these solutions are still prone to low-level bugs. Some parsers like GAPA [11] especially focus on safety or use a memory-safe language. As many errors still have to be handled correctly at runtime, these approaches are not sufficient for highly critical applications.

Lastly, parsers generated for interpreted languages [1,27], which rely on a complex runtime, have limited use for resource-constraint systems.

Summary In summary, we observe that no current solution offers expressiveness and legibility, combined with an easy venue towards formal verification and convincingly efficient generated code. None of the analyzed approaches is expressive enough to parse binary messages of existing real-world protocols including all its properties, ensures absence of runtime errors, and is suitable for embedded systems at the same time. Its design and implementation hence remains an open challenge.

3 Modeling and Processing Message Formats

In this section we introduce a methodology for the specification of message formats and subsequent generation of the corresponding verifiable parsers. Using Ethernet frames as a running example demonstrates several intricacies that require consideration. We start with a simple variant of an Ethernet frame and refine this definition iteratively to reach a complete specification. We set out to define the specification as a linear list of fields, like several previous approaches, but turn to a graph-based modeling later, to allow for strict specification of ambiguities in the standards. We then describe the algorithms to generate the code of the verifiable parsers. Each parser comprises a number of functions to validate and access the content of a message. We use Isabelle/HOL to formalize our model and describe the corresponding algorithms[3].

3.1 Example: Ethernet Frame

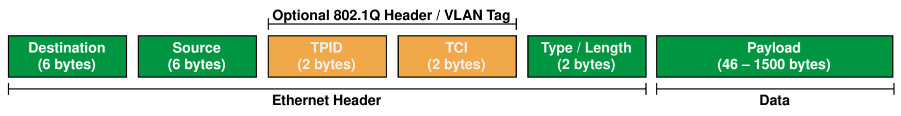

Fig. 1. Ethernet frame structure

Figure 1 depicts the basic structure of an Ethernet frame. Several variants of Ethernet exist, Ethernet II is most commonly used. It consists of two address fields of 6 bytes each, a Type field of 2 bytes encoding which protocol is encapsulated in the payload, and a variable-length Payload field which comprises the rest of the message. Both IEEE 802.3 and Ethernet II frames are used in practice,

[3] Isabelle is an automated proof assistant, HOL can be used as a functional programming language that allows proving certain properties. We refer the interested reader to [24] for an introduction into the matter.

so protocol instances have to distinguish them implicitly on the fly. The payload size is limited to 1500 bytes in Ethernet, so the field following the Source address is interpreted as IEEE 802.3 Length for values below 1500, and Ethernet II Type if the value is 1536 or above. Ethernet also defines extensions, like VLAN tagging (IEEE 802.1Q). It inserts a VLAN tag between the Source address and the Type/Length field, which consists of two fields: the TPID field and the TCI field. To determine if this extension is used, the instance checks the same field for the value 8100_{16}, and interprets the bytes as the TPID, the subsequent two bytes as control information, and only the subsequent bytes as Type or Length.

3.2 Message Representation

To generate a parser and reason about a message format like an Ethernet frame we need a formal specification that captures the message structure and all relevant constraints that must be enforced. The simplest possible representation of a message is a list of fields. For the Ethernet header each field can be represented by an identifier and a fixed length. The protocol may transmit Payload of different sizes, so a variable length value is needed for this part. To make matters worse, the payload length is defined depending on the overall length of the message in Ethernet II, using a mathematical expression. The underlying assumption is that there is a message buffer which comprises a number of bytes and potentially contains the message to parse. We use a deep embedding in our model for the representation (cmp. Appendix A):

datatype *'a field = Field 'a 'a expr*

Enumeration types can be used for identifiers, for instance for Ethernet II:

datatype *ethernet-v2 = Destination | Source | Type | Payload*

The message structure then is described as a list of fields. For the Destination, Source and Type field a fixed number is sufficient to specify the field length, while for the Payload field the length of the header needs to be subtracted from the length of the message. In our deep embedding, the variable *MessageLength* refers to the length of the message buffer. For sake of generality we define all lengths in bits and thereby enable the definition of non-byte-granular fields. An Ethernet II frame is thus defined as follows:

definition *ethernet-v2-frame :: ethernet-v2 field list* **where**
ethernet-v2-frame = [Field Destination (Num 48), Field Source (Num 48), Field Type (Num 16), Field Payload (Sub MessageLength (Num 112))]

To represent length fields like in the original IEEE 802.3 frame format, we need references to values of other fields in mathematical expressions. Our deep embedding contains the constructor *FieldValue 'a* for this purpose. Since our model works on bit granularity, but the Length field is byte-based, the value of the Length field has to be multiplied by 8. The specification for an IEEE 802.3 frame is as follows:

definition *ethernet-frame* :: *ethernet field list* **where**
ethernet-frame = [*Field Destination* (*Num 48*), *Field Source* (*Num 48*), *Field Length* (*Num 16*), *Field Payload* (*Mul* (*FieldValue Length*) (*Num 8*))]

To deal with the parallel use of both Ethernet frames in practice, either both formats could be treated as completely separate and handle their parallel use by other code. We decide to combine both variations in the same model instead. This increases model complexity slightly but prevents the need for manual handling of message variations, which could induce errors. Combining both formats prevents representation as a linear list of fields, as the semantics of the Payload Length depends on the value of the Type/Length field.

We hence have to allow for case distinctions. We described cases by a condition, which refers to the value of Type/Length, and a corresponding length expression for the Payload field. In other words, there are two distinct connections between the Type/Length and the Payload field. When modeling the dependencies between fields such connections can be interpreted as directed edges that define the order of the fields. Such edges contain two attributes: a condition that defines when the target field follows the source field and a length expression that specifies the length of the target field. Consequently, such edges are characterized by a source field, a target field and its attributes. A complete message thus forms a directed-acyclic graph (DAG) where nodes represent fields. Figure 4 shows the graph for the full specification of Ethernet frames.

We define conditions to be handled as boolean expressions, using the same deep embedding as used for mathematical expressions. Expressions must only contain references to field values of preceding nodes. This restriction prevents cyclic dependencies between expressions and ensures that a sequential evaluation of the validity of a message is possible.

Allowing for VLAN tagging further complicates specification. The existence of the 802.1Q header can only be determined by reading the two bytes following the Source address field. To resolve the ambiguity of potential fields we add a virtual node, *Type-Length-TPID* after the *Source*. It is solely used to differentiate the two message formats. It is followed by both *Type-Length* as well as *TPID*. We add a location expression at each edge that defines the position of the first bit of each respective field to be able to deal with the conditional overlay of fields in the model. For the specification of the field location our expressions allow using *FieldFirst* and *FieldLength* to refer to the location and the length of a previous field, respectively. We hence define edges by the following variant type, which is composed of source node, target node, condition, length expression, and location expression.

datatype *'a edge* =
  Edge 'a 'a 'a expr 'a expr 'a expr

Finally, we want to be able to describe the restriction on the payload length, and hence the overall message length, as an invariant. We hence introduce a final node that marks the end of the message, and as it is not followed by any other node it refers to the preceding nodes in the model, only. We introduce an initial node that defines the beginning of a message similarly, to define the length of the first field.

At this point we are able to fully define an Ethernet frame including all its variants. An excerpt of the full specification of Ethernet is shown below. An unabridged version can be found in Appendix B.

```
datatype ethernet-node = Init | Source | Destination | Type-Length-TPID | Type |
Payload | TPID | TCI | Final

definition ethernet-graph :: ethernet-node edge list where
ethernet-graph = [
  Edge Init Destination True (Num 48) (Num 0),
  Edge Destination Source True (Num 48) (Add (FieldFirst Destination) (FieldLength
Destination)),
 Edge Source Type-Length-TPID True (Num 16) (Add (FieldFirst Source) (FieldLength
Source)),
  Edge Type-Length-TPID Type (Ne (FieldValue Type-Length-TPID) (Num 0x8100))
(Num 16) (FieldFirst Type-Length-TPID),
  Edge Type-Length-TPID TPID (Eq (FieldValue Type-Length-TPID) (Num 0x8100))
(Num 16) (FieldFirst Type-Length-TPID),
  ...
]
```

We now turn to describing the functions that are generated for the parsers corresponding to our specification.

3.3 Derivation of Validation and Accessor Functions

Parsers are called with a given message as input, and have to extract the content, as specified above. They need to implement validation and accessor functions for each field, which we model as follows. A parser $\mathcal{P}$ consists of a list of validation and accessor functions. The validation function allows checking if all conditions stated in the specification hold for the message field. If this is the case, the corresponding accessor function can safely be used to retrieve the value of the field. For each message field the code for a validation function (*FieldValidFunc*) and the respective accessor function (*FieldAccessFunc*) have to be generated. In this manner all fields of a message can be validated and accessed consecutively.

Several conditions must hold for a bit array to be a valid message. First of all, a message field can only be valid if its first and last bit are within the range of the message buffer. Data out of range indicates an incomplete message. The validity of a message field depends on the specified conditions and the validity of its predecessor. Mapped on the model, this means that one incoming edge must have a valid condition and a valid source (except for the initial node) and, as the conditions of the outgoing edges can constrain the allowed values or the length of the field, the conditions of at least one outgoing edge must be fulfilled. Each path from the initial node to the final node denotes a variant of a message. A whole message is accepted if there is exactly one valid path.

Each node has to be reachable via at least one path from the initial node. The location of a field can vary because of optional or inserted fields and thus depends on a concrete path. Therefore the path has to be known to be able to calculate the location of a field. As conditions can refer to other fields, the path is also needed to evaluate a condition. For this reason, before we can validate or access a field, we have to determine all possible variants of a message, and for each variant the actual conditions and field bounds.

In the following we describe the algorithms for determining path attributes, variant functions, node paths, and field functions. We aim for simplicity in our algorithms. As the parser generation is only done once and the graphs which we use to represent message formats are rather small, the performance of the algorithms is not critical.

Path Attributes As one of the first steps of the parser generation the *path-attrs* algorithm derives the attributes for all possible paths from the initial node to any node in the graph. All references to other fields in expressions are eliminated during this process. The starting point of the algorithm is a graph definition, where each edge can be uniquely identified by an index number, e.g., by enumerating the edges of the graph definition.

```
type-synonym 'a agraph = (nat × 'a edge) list
```

The result of the algorithm is a list of tuples. For each path, which is represented by a list of indices, an expression for the condition, the length and the location of the first bit is returned.

The algorithm iterates over all edges of the graph definition. For each edge it determines all paths from the initial node by using the *paths* function. Applying *concat* on the resulting list of lists gives us a list of all paths from the initial node to any other node in the graph. Each path in this list is converted into the corresponding list of edges on the path by the *path-edges* function. From each list of edges the last edge is taken, and for this edge the condition, length and location expression extracted. References to other nodes in these expressions are replaced by the corresponding expression of the referenced node. This is realized by *subs* which recursively looks up the concrete expression in the graph definition.

```
definition path-attrs :: 'a agraph ⇒ (nat list × 'a expr × 'a expr × 'a expr) list where
path-attrs graph =
  [let edges = path-edges graph path in
     (path, subs edges (get-condition (last edges)),
      subs edges (get-length (last edges)),
      subs edges (get-first (last edges)))
   . path ← concat [paths graph i . i ← map fst graph]]
```

Variant Functions The parser $\mathcal{P}$ contains a variant validation function *VariantValidFunc* and a variant accessor function *VariantAccessFunc* for each message variant to allow the validation and access of a concrete variant of a field. These variant functions form the building blocks of the field validation functions and the field accessor functions. For each tuple generated by *path-attrs* containing condition, length and location for a specific path, a *VariantValidFunc* and a *VariantAccessFunc* are derived.

The body of a *VariantValidFunc* is based on the condition, a check which ensures that the field is within the bounds of the input buffer, and a call to the validation function of the preceding field, if it is not the first field of the message. Each variant function is identified by a path. As a path is represented by a list of indices, the preceding field can be determined by removing the last element of the current path.[4] Calls to other variant functions are denoted by *VariantValidCall* and *VariantAccessCall*, respectively.

```
fun variant-valid-funcs :: (nat list × 'a expr × 'a expr × 'a expr) list ⇒ 'a func list
where
variant-valid-funcs [] = [] |
variant-valid-funcs ((path, cond, len, first) # xs) =
  VariantValidFunc path (And (if init path ≠ [] then VariantValidCall (init path)
                                                  else True)
                             (And (Ge BufferLength (Add first len)) cond))
  # variant-valid-funcs xs
```

A *VariantAccessFunc* is defined by the location expression and the length expression of the field.

```
fun variant-access-funcs :: (nat list × 'a expr × 'a expr × 'a expr) list ⇒ 'a func list
where
variant-access-funcs [] = [] |
variant-access-funcs ((path, -, len, first) # xs) =
  VariantAccessFunc path (Value first len) # variant-access-funcs xs
```

The result of *variant-valid-funcs* and *variant-access-funcs* form the variant functions $\mathcal{V}$.

Node Paths The *node-paths* algorithm determines which paths lead to a field, i.e., which variants of a field exist, and which conditions at outgoing edges a node has. As described in Section 3.2 the values of a field can be further restricted by outgoing edges. At least one condition of an outgoing edge has to be fulfilled. Therefore, the corresponding conditions have to be determined as well. Like before, all references to other fields need to be resolved in dependence of a variant.

node-paths iterates over all nodes of the graph. The list of nodes of a graph is provided by *graph-nodes*. For each node all incoming edges are determined by *incoming*. Each incoming edge is used to determine all paths from the initial node by *paths*. For each path it then creates a tuple with two elements: the path represented by a list of indices and a disjunction of all conditions at outgoing edges. The disjunction is created by *any*, which takes a list of conditions from *path-conds*. *path-conds* extracts the conditions of the list of outgoing edges determined by *path-edges* and *outgoing*. Finally, the list of tuples is assigned to the corresponding node identifier.

```
definition node-paths :: 'a agraph ⇒ ('a × (nat list × 'a expr) list) list where
node-paths graph =
  [(node,
    concat [[(path,
              subs (path-edges graph path)
                   (any (path-conds (path-edges graph (outgoing graph node)))))
             . path ← paths graph edge]
            . edge ← incoming graph node])
   . node ← graph-nodes graph {}]
```

[4] The *init* function returns a list without its last element.

Field Functions A *FieldValidFunc* determines if one variant is valid. If a valid variant exists, the *FieldAccessFunc* can be used to return the value of the field. Field functions rely on the functionality provided by variant functions.

The resulting list of *node-paths* is used to generate validation and accessor functions for each field of the message. Each element of this list contains all the necessary information to create a validation and accessor function for one field. A field function is identified by a node identifier.

The algorithm *field-valid-funcs* creates a list of *FieldValidFunc*. In order that a field is valid, a variant of the field and the conditions of one of the outgoing edges must be valid. Hence, the body of a *FieldValidFunc* is a disjunction of calls to all variant validation functions and the corresponding expression which was determined for the conditions at outgoing edges.

```
fun field-valid-funcs :: ('a × (nat list × 'a expr) list) list ⇒ 'a func list where
field-valid-funcs [] = [] |
field-valid-funcs ((path, path-cond) # xs) =
  FieldValidFunc path (valid-calls path-cond) # field-valid-funcs xs
```

The body of each function is determined by *valid-calls*. *valid-calls* iterates over all path-condition tuples which it receives as arguments. For each path it creates a call to a *VariantValidFunc* and combines this call with the corresponding expression derived from the outgoing edges by a conjunction, as a variant is only valid if one of the conditions at the outgoing edges is valid. All created conjunctions are connected by a disjunction, as only one variant has to be valid.

```
fun valid-calls :: (nat list × 'a expr) list ⇒ 'a expr where
valid-calls [] = True |
valid-calls ((path, out-cond) # []) = And (VariantValidCall path) out-cond |
valid-calls ((path, out-cond) # xs) = Or (And (VariantValidCall path) out-cond)
                                         (valid-calls xs)
```

The list of field accessor functions is created by *field-access-funcs*. A *FieldAccessFunc* checks subsequently which variant of a field is valid and calls the corresponding *VariantAccessFunc*.

```
fun field-access-funcs :: ('a × (nat list × 'a expr) list) list ⇒ 'a func list where
field-access-funcs [] = [] |
field-access-funcs ((path, path-cond) # xs) =
  FieldAccessFunc path (access-calls path-cond) # field-access-funcs xs
```

The body of such a function is created by *access-calls*. It iterates over the list of paths and creates a nested if-expression, where the else-branch is created recursively. Each if-expression has a call to a *VariantValidFunc* as condition and a call to the corresponding *VariantAccessFunc* as body.

```
fun access-calls :: (nat list × 'a expr) list ⇒ 'a expr where
access-calls [] = Null |
access-calls ((path, -) # xs) =
  IfThenElse (VariantValidCall path) (VariantAccessCall path) (access-calls xs)
```

The result of *field-valid-funcs* and *field-access-funcs* form the field functions $\mathcal{F}$. The parser $\mathcal{P}$ comprises all variant functions $\mathcal{V}$ and field functions $\mathcal{F}$.

3.4 Message Refinement

Communication protocols are typically structured in layers. A protocol message contains a message of a higher layer protocol as its payload. We model this relation by message refinements. A message refinement is a tuple consisting of an identifier of the message, the name of the payload field, an identifier for the contained message and an expression. The expression describes under which conditions a message is contained in the payload field of another message. In the case of Ethernet the expression could specify that an IPv4 packet is contained in the Ethernet frame's payload field, if the `Type/Length` field has the value 0800_{16}. For each message multiple message refinements can be defined.

4 Implementation

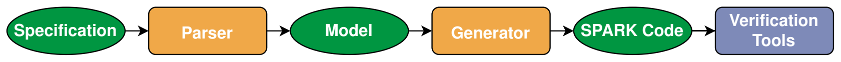

Fig. 2. Architecture

The RecordFlux toolset[5] comprises multiple parts (Figure 2). The specification language allows describing message formats and the relation of a message field to messages of higher protocol layers. The specification parser transforms this textual description into the model introduced in Section 3, which is used by the code generator. We chose SPARK [7] as the target language for code generation, as it already provides simple verification including all required tools. It is supported by the standard GCC toolchain and suitable for resource constrained systems.[6] We hence generate SPARK code, including all necessary function contracts. We then use the SPARK verification toolset to ensure the absence of runtime errors and the functional correctness of the generated code.

4.1 Specification Language

To specify message formats in a simple and readable manner, we have designed a specification language that allows expressing all properties of a message in accordance to our model. The specification language describes messages based on types. A type definition has the form: `type NAME is DEFINITION;`

The language supports two integer types to represent numbers: modular and range integers. A modular type represents the values from zero to one less than the modulus. The bit size of a modular type is determined by calculating the binary logarithm of the modulus. The destination and source address fields of Ethernet is represented by the following modular integer:

[5] RecordFlux is available as open source [5].
[6] We refer the interested reader [18] for an introduction into the language including all of its beneficial properties.

```
type Address is mod 2**48;
```

A range integer allows restricting the range of numbers by bounds. The set of values of a range type consists of all numbers from the lower bound to the upper bound. For a range type the bit size has to be specified explicitly. A range integer can be used for the Type/Length field, and allows incorporating the minimum length restriction of the payload field into the type definition:

```
type Type_Length is range 46 .. 2**16 - 1 with Size => 16;
```

This defines a type with a size of 16 bit which comprises all numbers from 46 to $2^{16} - 1$.

A message format is specified by a message type. A message type is a collection of components. Each component corresponds to one field in a message and is of form: `FIELD_NAME : FIELD_TYPE`. A simplified specification of an Ethernet II frame is as follows:

```
type Simplified_Frame is
   message
      Destination : Address;
      Source : Address;
      Type_Length : Type_Length;
      Payload : Payload;
   end message;
```

But as argued in Section 3 such a simple specification is not sufficient for Ethernet in general.

A then clause following a component allows defining which field follows. If no then clause is given, it is assumed that always the next component of the message follows. If no further component follows, it is assumed that the message ends with this field. A then clause can contain a condition under which the corresponding field follows and aspects which allow defining the length of the next field and the location of its first bit. The condition can refer to previous fields (including the component containing the then clause). In case of Ethernet two then clauses can be added to the Type/Length field to differentiate the two different meanings of this field:

```
   Type_Length : Type_Length
      then Payload
         with Length => Type_Length * 8
         if Type_Length <= 1500,
      then Payload
         with Length => Message'Last - Type_Length'Last
         if Type_Length >= 1536;
```

The full specification of an Ethernet frame including VLAN tags is shown in Figure 3. Figure 4 depicts the corresponding graph representation. The package `Ethernet` consists of multiple integer types and a message type `Frame`. Packages are used to structure a specification and thus make the specification modular.

A type refinement describes the relation of a component in a message type to another message type. It states under which condition a specific protocol message is expected inside of a payload field. Only components of the built-in

```
package Ethernet is

   type Address is mod 2**48;
   type Type_Length is range 46 .. 2**16 - 1
      with Size => 16;
   type TPID is range 16#8100# .. 16#8100#
      with Size => 16;
   type TCI is mod 2**16;

   type Frame is
     message
       Destination : Address;
       Source : Address;
       Type_Length_TPID : Type_Length
          then TPID
             with First => Type_Length_TPID'First
             if Type_Length_TPID = 16#8100#,
          then Type_Length
             with First => Type_Length_TPID'First
             if Type_Length_TPID /= 16#8100#;
       TPID : TPID;
       TCI : TCI;
       Type_Length : Type_Length
          then Payload
             with Length => Type_Length * 8
             if Type_Length <= 1500,
          then Payload
             with Length => Message'Last - Type_Length'Last
             if Type_Length >= 1536;
       Payload : Payload
          then null
             if Payload'Length / 8 >= 46
                 and Payload'Length / 8 <= 1500;
     end message;

end Ethernet;
```

Fig. 3. Full specification of an Ethernet frame covering Ethernet II, IEEE 802.3, and IEEE 802.1Q

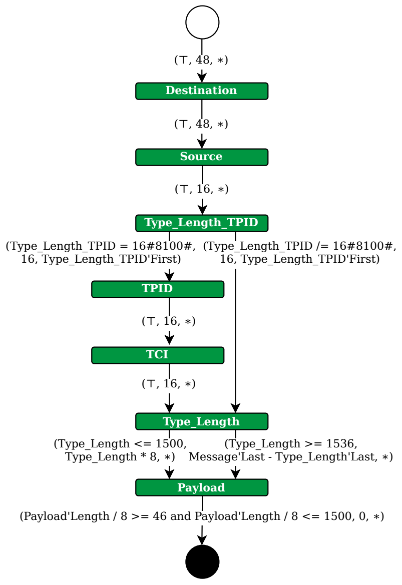
(⊤, 48, ∗)
Destination
(⊤, 48, ∗)
Source
(⊤, 16, ∗)
Type_Length_TPID
(Type_Length_TPID = 16#8100#, 16, Type_Length_TPID'First)
(Type_Length_TPID /= 16#8100#, 16, Type_Length_TPID'First)
TPID
(⊤, 16, ∗)
TCI
(⊤, 16, ∗)
Type_Length
(Type_Length <= 1500, Type_Length * 8, ∗)
(Type_Length >= 1536, Message'Last - Type_Length'Last, ∗)
Payload
(Payload'Length / 8 >= 46 and Payload'Length / 8 <= 1500, 0, ∗)

Fig. 4. Graph representation of Ethernet frame specification (Notation: For an edge $e = (s,t,c,l,f)$: $*$ denotes `f = s'First + s'Length`, $\top$ denotes `c = True`)

type `Payload` can be refined. Types defined in other packages are referenced by a qualified name in the form `package_name.message_type_name`. The condition can refer to components of the refined type.

```
type IPv4_In_Ethernet is new Ethernet.Frame (Payload => IPv4.Packet)
   if Type_Length = 16#0800#;
```

In this example the relation between an Ethernet frame and an IPv4 packet is specified. The message type `Frame` in package `Ethernet` contains a `Packet` defined in package `IPv4` if the `Type_Length` field of the Ethernet frame equals to `0x0800`.

4.2 Code Generation

The basis for the code generation is the model described in Section 3. The generated code takes a plain byte array as input and allows validating and accessing the message data in a structured way. For each specified message a number of functions is generated. The user of the generated code finds a validation function and accessor function for each field of the message. The validity of a field must be checked before accessing its value. This is realized by preconditions. By this means it is ensured that the value of a field is only accessible if all previous fields and the value of the field is valid.

Applying a function to a wrong input buffer or to an incorrect part of the buffer could lead to unexpected results. To prevent this a buffer has to be labeled correctly. This is realized by a predicate used as precondition of all validation and accessor functions. A label is added automatically if the relation between a payload field and a contained message is specified by a type refinement, and a contains function is used to check if the corresponding conditions are fulfilled for the input buffer in question. A contains function is the representation of a type refinement in the generated code. If the input data is received from an external source, the input buffer must be labeled explicitly.

The structure of the specification is reflected in the generated code. As a result it is possible to keep the code as well as the specification modular and extendable. For example type refinements can be defined in a separate specification. This allows adding further higher layer protocols transmitted in an already specified protocol without changing existing code.

4.3 Verification

The SPARK programming language allows the detailed specification of the behavior of software components by the use of contracts. This specification is used by the SPARK verification tools to formally proof that the stated properties of the program hold. The achievable assurance ranges from showing that no runtime exceptions occur to ensuring functional correctness based on a formal specification. This is realized by analyzing the source code and generating verification conditions, which are then passed to multiple theorem provers to formally verify the correctness of the code.

The use of SPARK allows us proving the absence of runtime errors and the correct use of the generated code. All of the generated code is valid SPARK code and will be analyzed by the verification tools. The incorrect use of the generated code, e.g. accessing a field value without prior verification, is prevented by adding appropriate contracts to these functions.

A key benefit of using SPARK is that the code generator need not to be trusted with regard to the absence of runtime errors in the generated code, as this property is proved by the verification tools. Furthermore, the SPARK verification tools assist ensuring the correctness of the specified message format. For example the tools will find potential integer overflows in expressions, which could indicate a missing restriction of the value range of a field.

5 Case Studies

We demonstrate the applicability of RecordFlux using the example of TLS in two case studies. In the first study we replaced the whole parsing code of an existing TLS library by an implementation specified and generated by RecordFlux and analyzed its impact. In the second study we used RecordFlux to specify and parse messages of the TLS Heartbeat protocol, an optional extension of the TLS standard, which is not supported by the library used in the first study.

5.1 Verified TLS Parser

Fizz [22] is a TLS 1.3 implementation developed and used by Facebook, written in C++. As a proof of concept we have replaced the C++ parser by verified SPARK code. Therefore, we used our specification language (see Section 4.1) to specify the messages of TLS 1.3, as standardized in RFC8446. Based on this specification, RecordFlux generated SPARK code for TLS Record messages, TLS Handshake messages and TLS extensions. We integrated the existing C++ code with the generated SPARK code manually. The glue code mainly performs the conversion between C++-specific structures like vectors and SPARK-compatible data formats.

Security Parsing of protocol messages is a sensitive part of a TLS implementation. [9] reports an integer overflow in Fizz. An exploit could have left the application using Fizz in an infinite loop, just by sending a short sequence of messages with a well-chosen value in the length field of a TLS Record message. Facebook fixed the bug by choosing a bigger integer type (`size_t` instead of `uint16`). RecordFlux checks the length field for allowed values before continuing the parsing, which we argue is a better solution to the problem. The SPARK verification tools can then prove the absence of unexpected integer overflows.

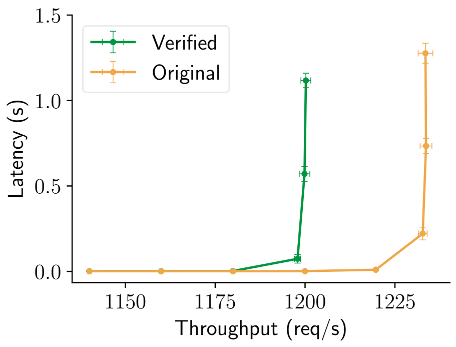

Fig. 5. Performance at Handshake layer; separate TLS handshake for each request

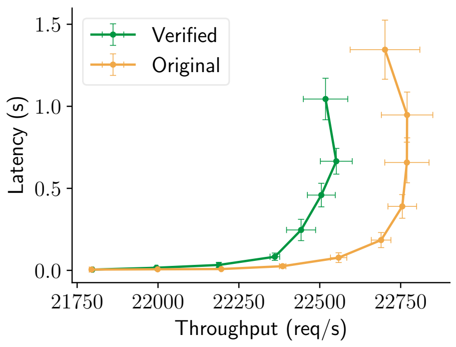

Fig. 6. Performance at Record layer; one TLS handshake for multiple requests

Performance Performance is considered at least of equal importance as security in practice. We hence evaluated the performance impact of replacing the original message parser with the code generated by RecordFlux. For that purpose we used a modified version of wrk2 [6], where we added the possibility to run in two different modes. To measure the impact of RecordFlux on the TLS handshake, the first mode of wrk2 creates a TLS connection for each HTTP request. The impact of RecordFlux during data transmission is measured by the second mode of wrk2 that only creates one TLS connection before sending requests. wrk2 sends requests in a constant rate and measures the resulting throughput and latency of the responses. The sending rate of wrk2 is increased iteratively until the throughput is not improving any more. For each sending rate the mean values of 40 measurements with a duration of 60 seconds each are calculated. To minimize the impact of network hardware on the results we run Fizz and wrk2 on the same machine.

We expected to see some performance impact due to the additional validation checks in the generated code and the conversions between C++ and SPARK structures. The diagrams in Figures 5 and 6 show the resulting mean values of throughput and latency and the corresponding 95 % confidence intervals. The maximum throughput is around 2.7 % lower in the Handshake layer and 1.1 % lower in the Record layer compared to the original parser. An analysis of the CPU cycles used by both variants with Valgrind [23] showed that the majority of additional cycles are spent on memory allocations and processing of data conversions in the glue code.

The results show that there is no significant performance degradation. We conclude that the approach is generally applicable, although mixing existing C++ code with SPARK code is not ideal from the point of view of performance.

5.2 TLS Heartbeat

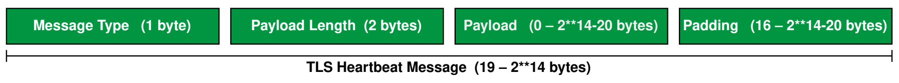
Message Type (1 byte)
Payload Length (2 bytes)
Payload (0 – 2**14-20 bytes)
Padding (16 – 2**14-20 bytes)
TLS Heartbeat Message (19 – 2**14 bytes)

Fig. 7. Message format of a TLS Heartbeat

The Heartbeat extension adds keep-alive functionality to TLS. It gained inglorious prominence by Heartbleed [19], a security vulnerability in the OpenSSL library that affected millions of devices. Heartbleed allowed to extract sensitive data from a TLS endpoint due to an improper input validation.

Both sides of a TLS connection can request the use of the Heartbeat protocol during the TLS handshake. If accepted by the other side, the initiator is allowed to periodically send Heartbeat requests during the lifetime of the TLS connection. Each request contains payload of arbitrary length and content. The receiver of a Heartbeat request must send a response back which contains the same payload as the request. The format of a TLS Heartbeat message is shown in Figure 7. The corresponding RecordFlux specification of a TLS Heartbeat message is as follows:

```
package TLS_Heartbeat is

   type Message_Type is (HEARTBEAT_REQUEST => 1, HEARTBEAT_RESPONSE => 2) with Size => 8;
   type Length is range 0 .. 2**14 - 20 with Size => 16;

   type Heartbeat_Message is
      message
         Message_Type : Message_Type;
         Payload_Length : Length
            then Payload with Length = Payload_Length * 8;
         Payload : Payload
            then Padding with Length = Message'Last - Payload'Last;
         Padding : Payload
            then null if Message'Length <= 2**14 * 8 and Padding'Length >= 16 * 8;
      end message;

end TLS_Heartbeat;
```

Heartbeat_Message represents a TLS Heartbeat Message. Such a message consists of four fields: The Message_Type field specifies the type of the message. It is represented by a enumeration type with a size of 1 byte and comprises two valid values: 1 for a request and 2 for a response. All other values are considered invalid. Payload_Length defines the length of the following Payload field. The Payload field contains the content of the message, and Padding comprises the rest of the message.

The lengths of the Payload and the Padding field is explicitly defined by length expressions. The whole message is restricted to a length of 2^{14} bytes. The Padding field must be at least 16 bytes long.

The following excerpt shows some of the generated SPARK subprogram declarations for the a Heartbeat_Message:

```
function Is_Contained (Buffer : Bytes) return Boolean with Ghost, Import;

procedure Label (Buffer : Bytes) with Ghost, Post => Is_Contained (Buffer);

function Valid_Message_Type (Buffer : Bytes) return Boolean
  with Pre => Is_Contained (Buffer);

function Get_Message_Type (Buffer : Bytes) return Message_Type
  with Pre => (Is_Contained (Buffer) and then Valid_Message_Type (Buffer));

function Valid_Payload (Buffer : Bytes) return Boolean
  with Pre => Is_Contained (Buffer);

procedure Get_Payload (Buffer : Bytes; First : out Natural; Last : out Natural)
  with Pre => (Is_Contained (Buffer) and then Valid_Payload (Buffer)),
       Post => (First = Get_Payload_First (Buffer) and then
                Last = Get_Payload_Last (Buffer));

function Is_Valid (Buffer : Bytes) return Boolean
  with Pre => Is_Contained (Buffer);
```

The Is_Contained function which is a precondition for all validation and accessor functions ensures that always the correct message buffer is used. The Is_Contained predicate is set automatically for all defined message refinements. If a message buffer is received from an external source it can be explicitly labeled with the Label function. For each message field a validation function (prefixed with Valid) and an accessor function (prefixed with Get) is created. Each accessor function has the corresponding validation function as a precondition. This ensures that the validity is always checked before accessing a field value. The `Is_Valid` checks the validity of the whole message. It returns `True` if the input buffer contains one valid message variant.

The following code example shows how the generated code is used:

```
with IO;
with TLS.Heartbeat_Message; use TLS.Heartbeat_Message;

procedure Main is
   Buffer : Bytes := IO.Read;
   Tag    : Message_Type;
   First  : Natural;
   Last   : Natural;
begin
   Label (Buffer);
   if Is_Valid (Buffer) then
      Tag := Get_Message_Type (Buffer);
      Get_Payload (Buffer, First, Last);
      Process_Payload (Buffer (First .. Last));
   end if;
end Main;
```

The message buffer is read from an external source. In this example the message buffer is explicitly labeled as a buffer which should contain a TLS Heartbeat message. By also specifying the Record layer with RecordFlux and defining a message refinement between Record message and Heartbeat message, no labeling would be needed. The validity of content of `Buffer` is checked by `Is_Valid`. Alternatively a user could also check the validity of each field on his own. As the `Padding` must be ignored and the relation between the `Payload_Length` field and the `Payload` field is internally known, a user only needs to access the `Message_Type` and the `Payload`.

The SPARK verification tools ensure the correct use of the generated code. If a user would not check the validity of the input buffer, the tools will find this mistake when proving the correctness of the code. `Get_Message_Type` and `Get_Payload` will be flagged with `precondition might fail`.

While the Heartbeat protocol appears quite simple, a flawed implementation can have serious implications. Heartbleed allowed to send a request with a high length value while sending just a short payload and padding. On the receiver side the length value was not checked against the actual received payload. This led to a buffer overflow, so that not only the payload of the request was sent back, but also data following the message buffer.

RecordFlux enforces that the length of a payload field is always defined by a length expression. The code generator adds checks to ensure that the value of a length field is in the allowed range and the message to parse is long enough, and so prevents the issue seen in Heartbleed. Even if RecordFlux would erroneously miss adding a necessary check, the SPARK verification tools will find the potential buffer-overflow before faulty code is used involuntarily.

6 Conclusion and Outlook

We have created a methodology for specification of message formats of communication protocols and the automatic generation of a parser. Based on this methodology we have created a practical implementation, which comprises a DSL for describing message formats and code generator that creates SPARK code, for which the absence of runtime errors can be shown. The generated code is applicable in real-world applications as demonstrated for TLS 1.3, which proved to suffer only from minor performance penalties despite its proven security.

So far we only handled the parsing of messages, but this is only one part of a protocol. In the future we will also look into the protocol logic. We aim to extend the current methodology to get a full formal specification of a protocol, to generate provable secure code.

Acknowledgements This work is partially funded by the European Union (EU), the European Social Fund (ESF), tax money on the basis of the budget approved by the members of the parliament of Saxony, and the Cluster of Excellence EXC 2050 “Centre for Tactile Internet with Human-in-the-Loop” (CeTI).

References

1. Construct: Declarative data structures for python that allow symmetric parsing and building, https://github.com/construct/construct, 2019
2. Hammer, https://github.com/UpstandingHackers/hammer, 2019
3. Kaitai Struct: declarative language to generate binary data parsers, https://github.com/kaitai-io/kaitai_struct, 2019
4. Protocol Buffers, https://developers.google.com/protocol-buffers/, 2019
5. RecordFlux, https://github.com/Componolit/RecordFlux, 2019
6. wrk2: A constant throughput, correct latency recording variant of wrk, https://github.com/treiher/wrk2, 2019
7. SPARK (2019), https://www.adacore.com/about-spark
8. Back, G.: DataScript - A Specification and Scripting Language for Binary Data. In: Goos, G., Hartmanis, J., van Leeuwen, J., Batory, D., Consel, C., Taha, W. (eds.) Generative Programming and Component Engineering, vol. 2487, pp. 66–77. Springer Berlin Heidelberg, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45821-2_4, http://link.springer.com/10.1007/3-540-45821-2_4
9. Backhouse, K.: Facebook fizz integer overflow vulnerability (CVE-2019-3560) (2019), https://blog.semmle.com/facebook-fizz-CVE-2019-3560/
10. Bangert, J., Zeldovich, N.: Nail: A practical tool for parsing and generating data formats. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). pp. 615–628. USENIX Association, Broomfield, CO (2014), https://www.usenix.org/conference/osdi14/technical-sessions/presentation/bangert
11. Borisov, N., Brumley, D., Wang, H.J., Dunagan, J., Joshi, P., Guo, C.: Generic Application-Level Protocol Analyzer and its Language. In: NDSS (2007)
12. Bratus, S., Patterson, M.L., Hirsch, D.: From shotgun parsers to more secure stacks. Shmoocon, Nov (2013)

13. Couprie, G.: Nom, a byte oriented, streaming, zero copy, parser combinators library in rust. In: 2015 IEEE Security and Privacy Workshops. pp. 142–148. IEEE (2015)
14. Daly, M., Mandelbaum, Y., Walker, D., Fernández, M., Fisher, K., Gruber, R., Zheng, X.: PADS: an end-to-end system for processing ad hoc data. In: Proceedings of the 2006 ACM SIGMOD international conference on Management of data - SIGMOD '06. p. 727. ACM Press, Chicago, IL, USA (2006). https://doi.org/10.1145/1142473.1142568, http://portal.acm.org/citation.cfm?doid=1142473.1142568
15. ITU-T: Recommendation X.680 - Abstract Syntax Notation One (ASN.1): Specification of basic notation (2015)
16. Lions, J.L., et al.: Flight 501 failure. Report by the Inquiry Board (1996)
17. McCann, P.J., Chandra, S.: Packet Types: Abstract Specification of Network Protocol Messages. In: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication. pp. 321–333. SIGCOMM '00, ACM, New York, NY, USA (2000). https://doi.org/10.1145/347059.347563, http://doi.acm.org/10.1145/347059.347563
18. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press (2015)
19. MITRE: CVE-2014-0160, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160
20. MITRE: CVE-2017-0144, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0144
21. MITRE: CVE-2017-14315, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-14315
22. Nekritz, K., Iyengar, S., Guzman, A.: Deploying TLS 1.3 at scale with Fizz, a performant open source TLS library (2018), https://code.fb.com/security/fizz/
23. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: ACM Sigplan notices. vol. 42, pp. 89–100. ACM (2007)
24. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002)
25. O'Sullivan, B.: Attoparsec, https://github.com/bos/attoparsec, 2019
26. Pang, R., Paxson, V., Sommer, R., Peterson, L.: Binpac: A Yacc for Writing Application Protocol Parsers. In: Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement. pp. 289–300. IMC '06, ACM, New York, NY, USA (2006). https://doi.org/10.1145/1177080.1177119, http://doi.acm.org/10.1145/1177080.1177119
27. Rodriguez, A., McGrath, R.E.: Daffodil: A new dfdl parser (2010)
28. Srinivasan, R.: XDR: External data representation standard. Tech. rep. (1995)

A Deep Embedding

```
datatype 'a expr =
  Num int |
  Add 'a expr 'a expr |
  Mul 'a expr 'a expr |
  Sub 'a expr 'a expr |
  Div 'a expr 'a expr |
  Value 'a expr 'a expr |
  FieldValue 'a |
  FieldLength 'a |
  FieldFirst 'a |
  MessageLength |
  MessageFirst |
  MessageLast |
  BufferLength |
  VariantValidCall nat list |
  VariantAccessCall nat list |
  Null |
  True |
  And 'a expr 'a expr |
  Or 'a expr 'a expr |
  Eq 'a expr 'a expr |
  Ne 'a expr 'a expr |
  Lt 'a expr 'a expr |
  Le 'a expr 'a expr |
  Gt 'a expr 'a expr |
  Ge 'a expr 'a expr |
  IfThenElse 'a expr 'a expr 'a expr
```

B Formal Specification of Ethernet Frame

```
definition ethernet-graph :: ethernet-node edge list where
ethernet-graph = [
  Edge Init Destination True (Num 48) (Num 0),
  Edge Destination Source True (Num 48) (Add (FieldFirst Destination) (FieldLength
Destination)),
  Edge Source Type-Length-TPID True (Num 16) (Add (FieldFirst Source) (FieldLength
Source)),
  Edge Type-Length-TPID Type (Ne (FieldValue Type-Length-TPID) (Num 0x8100))
(Num 16) (FieldFirst Type-Length-TPID),
  Edge Type-Length-TPID TPID (Eq (FieldValue Type-Length-TPID) (Num 0x8100))
(Num 16) (FieldFirst Type-Length-TPID),
  Edge TPID TCI True (Num 16) (Add (FieldFirst TPID) (FieldLength TPID)),
  Edge TCI Type True (Num 16) (Add (FieldFirst TCI) (FieldLength TCI)),
  Edge Type Payload (Le (FieldValue Type) (Num 1500)) (Mul (FieldValue Type) (Num
8)) (Add (FieldFirst Type) (FieldLength Type)),
  Edge Type Payload (Ge (FieldValue Type) (Num 1536)) (Add (Sub MessageLast (Add
(FieldFirst Type) (FieldLength Type))) (Num 1)) (Add (FieldFirst Type) (FieldLength
Type)),
  Edge Payload Final (And (Ge (FieldLength Payload) (Num 368)) (Le (FieldLength
Payload) (Num 12000))) (Num 0) (Add (FieldFirst Payload) (FieldLength Payload))
]
```