Context Sensitive and Secure Parser Generation for Deep Packet Inspection of Binary Protocols

by

Ali ElShakankiry

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
August 2017

Copyright © Ali ElShakankiry, 2017

Abstract

Network protocol parsers constantly dissect a large number of network data to place into internal data structures for further processing by traffic analysis systems. Many network protocol parsers are hand-written for performance reasons, and lack the security required to run on mission-critical networks. We propose an approach that automatically generates custom protocol parsers to process network traffic to be used as part of an Intrusion Detection System. The user is provided a specification language in which they can define the protocols they need to analyse. This thesis looks at command and control/industrial control networks that are characterized by a limited number of known protocols. We present a robust, secure, and high-performing solution that deals with the issues that have only partially been addressed in this domain.

Acknowledgments

I would like to thank Dr. Thomas R. Dean for the continued advice and support throughout the duration of my research at Queen's. The late afternoon lab talks on almost anything and everything have definitely broadened my world view and knowledge.

I would also like to dedicate this work to my family. This thesis would not have been possible without their constant support. I am evermore grateful for the opportunity to pursue my dreams, and to get the best possible education doing so.

Statement of Originality

I, Ali ElShakankiry, hereby declare that I am the sole author of this thesis. All ideas and inventions attributed to others have been properly referenced. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Network protocol parsers are used extensively in systems today to ensure traffic integrity and security. While essential for network intrusion detection systems (NIDS), firewalls and general network analyzers, many network protocol parsers are still written by hand and from scratch for their specific purpose. Writing custom network parsers by hand is time consuming, and prone to security vulnerabilities as can be seen from protocol analysis tools like Wireshark [12, 26]. We introduce a parser generator framework that produces high-performing parsers with deep packet inspection (DPI) capabilities. The goal of our project is to provide a parser generator framework that is meant to be used to fully generate an NIDS system that is to be used in limited networks as defined by Hasan et al. [16].

There have been multiple efforts in the research community to create general-purpose and network protocol parser generators, all with their own domain-specific languages and goals. Our parser generator framework is aimed at providing a domain-specific language with the ability to specify parsing constraints that closely resemble network protocol definitions in Request for Comments (RFCs). The framework also allows for the specification of intrusion detection constraints

all in the input language. The output of our framework generates custom protocol parsers in C that are context-dependent and have been used in our NIDS prototype to inspect ARP [29], IGMP [9], and multiple UDP [30] protocols.

1.1 Overall System Architecture

The network protocol parser generator is part of a larger joint project between Queens University, The Royal Military College both in Kingston, Ontario, and École Polytechnique de Montréal. The aim is to create a real-time Network Intrusion Detection System (NIDS) for limited networks where the protocols used are constrained to a known, established set of allowed traffic. Any traffic outside of this set is flagged as anomalous. It should be noted that deep packet inspection is possible in these circumstances since network traffic is controlled by the entities using them. Protocols that use encrypted communications can be set up to allow the data to be inspected only by an NIDS. The main protocol to be used in the system is the Data Distribution Service (DDS) protocol. The preliminary plan for the system architecture has been laid out, and partially implemented.

Figure 1.1 shows the overall architecture of the NIDS system. The protocol description is used to generate the needed analysis code for the parser, extractor, and constraint checker. The preliminary design included an extractor that provides the packet summary information given to the constraint engine. All of the system beginning at the parser up to and including the packet summary information section are considered as the network protocol parser in the thesis. The constraint checker is part of the semantic validation of the data, and attempts to establish a set of known traffic through a learning mode of the real network data. Constraints

are determined at the protocol level in the protocol descriptions and inter-packet constraints can be specified for the constraint checking. The parser deals with network data by analysing one packet at a time. The detection system's purpose is to attempt to recognize traffic that is malicious. The sole purpose of this system is to alert the network administrators that there is a possible intrusion. This system is not intended to block any traffic on its own. The input to the system are individual network packets, which are inspected. The only output of this system are network alerts. Network packets can be mirrored to be inspected on a modern switch or router, allowing for the possibility of packet inspection without modifying or preventing network packets from proceeding to their destination.

A previous parser was functional as an interpreter written in C++, and runs slower than the incoming traffic being tested for the NIDS. In order to function the parser requires the formal definitions of the protocols, or the defined grammar of the protocols. These grammars are included in the protocol description. The protocol description provides a modular interface where the grammars are described as encoded Extensible Markup Language (XML) documents. These documents are created through a source transformation from a defined protocol, the Syntax Constrained Language (SCL), using the TXL programming language. The parser uses the grammar graphs to structurally decode the binary packets of network traffic into a usable representation. The previous parser decodes the packets into text files that are written on disk, creating most of the unwanted bottleneck. While the grammars provided are also in XML files, they only need to be built internally once the system is running, removing any extra overhead every time a specific protocol is being parsed.

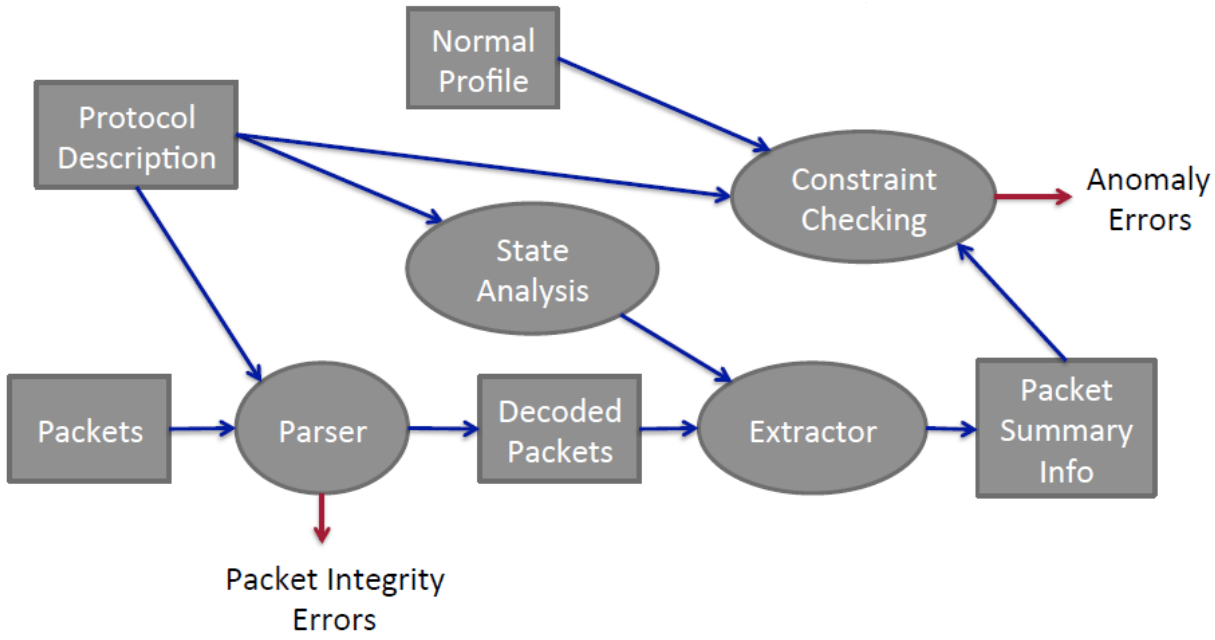

Figure 1.1: NIDS Architecture.

The parser is ultimately responsible for decoding the binary packets into a known readable structure, and flagging any packets for errors in its structure based on protocol constraints. The parser does not deal with semantic constraint checking. This is done later in the system as seen in Figure 1.1. Extensions to the parser may be done in order to optimize it and increase its throughput which is the main focus of this contribution.

1.2 Parser Requirements

A network-based intrusion detection system needs to ensure that there is no malicious network data. While it is not possible to achieve perfect intrusion detection, it is essential that an NIDS' parser operates with the following requirements in mind. The parser needs to support the maximum throughput of the network monitor. It

should be robust and run at all times without crashing. The parser itself needs to be secure to ensure that it does not introduce a new attack vector on the network. From the end user's perspective, the parser needs to be maintainable. One approach involves providing a high-level language specification with the ability to add and remove modules for protocols that function at different networking layers. We briefly discuss how our generated parser achieves these goals.

1.2.1 High-Performance

The generated parser provides deep-packet inspection of protocols while maintaining support for high-bandwidth networks. This is achieved by generating custom dissectors or parsers for every protocol that is defined by the user.

1.2.2 Reliability

Once parsed packets are no longer needed, all allocated memory is properly freed. This ensures that there are no memory leaks, and that the parser can run with no down-time. Any parse that fails will free the allocated memory and ignore the packet. Trace files can be used to debug parses that have failed.

1.2.3 Security

The remaining unparsed data is always referenced before a parse to ensure that the required length of the next data attribute is available. Pointers to optional data structures and lists of objects are always initialized. Parses will fail if there is remaining data in a packet that has not been dissected. This removes the possibility of parsing packets with unintended or malicious data.

1.2.4 High-Level Specification

ASN.1 notation was introduced by ITU [7, 18] to specify binary protocols. By extending ASN.1, Syntax Constraint Language [21] allows the user to specify constraints on how to parse a network protocol by utilizing notation that is created specifically for internet infrastructure protocols like OSPF [19] and SNMP [3]. We take advantage of this feature by extending SCL in our work.

1.2.5 Modular

Protocols that are defined in SCL may be used by other modules. This allows the grammars to be layered accordingly, and allows a single parser to be generated for all of the defined network protocols. The ASN.1 language by design has the concept of exporting objects and using them in other modules. Since we are extending SCL, we get the benefit of this feature.

1.3 Thesis Contribution

The thesis provides two main contributions to research in network traffic analysis. The first is the extensions that have been made to the SCL language. These extensions provide the basis for being able to use SCL as a specification language for binary protocol parsers, and for context-sensitive parsers in general. The second contribution is the parser generator framework itself, which uses a different technique than other protocol parser generators that are used for binary protocols today. It's aim is to keep the framework as simple as possible, while using a readable specification language. The working version of the parser generator framework is currently available on Github [8].

1.4 Thesis Organization

The structure of the following sections in the thesis is as follows. Chapter 2 provides a description of the backbone of our DPI framework. Chapters 3 and 4 describe the input and output to the parser generator framework in depth. The evaluation follows in Chapter 5. We end the thesis by concluding and laying out plans for future work in Chapter 6.

Chapter 2

Background

2.1 Terminology

Limited Network: In this context, a limited network is one in which all of the networking protocols that exist on the network are known and used for a special purpose. Examples of limited networks include air traffic control systems and factory networks. Limited networks are controlled by the entities that are using them, which define their own networking restrictions and security models. In this case, an NIDS can be tuned to properly filter out and flag abnormal data as defined by the networking and protocol restrictions implemented.

Domain Specific Language (DSL): A specification or programming language that is customized for a specific application area. In this project, we have extended a domain specific language, Syntax Constraint Language (SCL), to define TCP/IP networking protocols. These network protocol specifications in the Syntax Constraint Language act as the input to the parser generator framework and the NIDS system as a whole.

Grammar: The specification and definition of the syntax of a language. In this context, a domain specific language is given to the user to specify network protocols according to the grammar of that specification language. Generated parsers will also be created based on the grammar of the protocols defined in SCL.

Deep Packet Inspection (DPI): The filtering of data and payload sections of network packets in addition to packet headers. This is the main goal of our NIDS project, to be able to filter and analyse full network packet data to determine malicious data in live networks.

2.2 Environment

The target environment for the use of this NIDS system involves limited networks in which power usage and heat output from computing devices should be kept to a minimum. Due to these constraints, the goal is to reduce the computing requirements, and to optimize the algorithm used to parse the network data. For this reason, the goal of the project is to maximize the performance of the parsers without introducing a multi-processor solution. The current environment used to test the parser generator framework is a dual-core CPU, where the parser runs as a single process on a single core.

2.3 The TXL Source Transformation Language

TXL is a functional programming language used for source to source transformations and rapid prototyping in multiple software engineering applications [4, 5]. It has been used in solving large-scale real world problems involving billions of

lines of code such as the year 2000 problem [6]. The language comprises of two stages in the process of creating a source transformation. The first stage is defining context-free grammars for the languages that a user is translating. A parser for the grammars defined are derived by the TXL engine and are not modifiable by the user. These grammars can be extended or overridden by the user for certain instances of transformations. The second is a set of by-example source transformations, comprised of rules and functions that match contextual information of the original source language, and transform them to the target language.

We have extensively used TXL as the backbone for generating our binary protocol parsers. TXL is used for multiple stages in our parser generator framework. We begin by ensuring our input language syntax and part of the semantic information is correct as provided by the user, and end at generating our output source code. The output is C source code that carries out the DPI on binary protocols. The output format closely resembles hand written protocol parsers.

2.4 Syntax Constraint Language

SCL provides the network security specialist a means to define a modular syntax specification for a binary protocol. SCL is an extension to ASN.1, and is used due to its ease of defining network protocols particularly among network specialists. The extensions developed by Marquis et al. [21] introduce blocks of XML markup for sequences in ASN.1 that allow constraints to be specified. By using SCL constraints, we can embed the semantic information of an object's attributes alongside their syntactic specification. This extends ASN.1 to allow the definition of context-sensitive grammars. SCL has been used before to test the security of network

protocol data by mutating packets and reassembling them to wire format data [1, 35].

In the next chapter we discuss the importance of adapting ASN.1 and the old SCL language to create a parser that is modular. In addition to modularity, we discuss the constraints added by the SCL specification to allow context-sensitive parsing.

2.5 ASN.1

Figure 2.1 shows an NTPV4 module defined in SCL that exports its top level object. In this case, the full protocol data unit (PDU) for the Network Time Protocol [24]. Exporting PDU from the NTPV4 module allows the UDP module in Figure 2.2 to import and define parser entry points for all of the protocols that run on top of UDP. This is done by importing all of the known modules that use UDP to send network data. Note that the UDP module example in Figure 2.2 will not generate any layer 4 parsing for the UDP protocol. In this case, an IDS developer must manually skip the UDP header at layer 4 to pass the data to the generated parser, which will attempt to parse an NTPV4 or RTPS packet. The generated parser for this module expects to be called and passed the PDU of the child protocols without any of the UDP header information.

By leveraging the power of multiple modules and layering the protocols in ASN.1, we can add a full UDP module definition as seen in Figure 2.3. This now allows a parser to be generated that will begin parsing at layer 4. The user in this case is then responsible for passing the data from the packet at the beginning of the UDP header. This example can be extended to generate a parser that begins at layer

```
1  NTPV4 DEFINITIONS ::= BEGIN
2
3     EXPORTS PDU;
4
5  PDU ::= SEQUENCE {
6     flags            INTEGER (SIZE 1 BYTES),
7     peerStratum      INTEGER (SIZE 1 BYTES),
8     peerInterval     INTEGER (SIZE 1 BYTES),
9     peerPrecision    INTEGER (SIZE 1 BYTES),
10    rootDelay        INTEGER (SIZE 4 BYTES),
11    rootDispersion   INTEGER (SIZE 4 BYTES),
12    referenceId      INTEGER (SIZE 4 BYTES),
13    referenceTS      INTEGER (SIZE 8 BYTES),
14    originTS         INTEGER (SIZE 8 BYTES),
15    recieveTS        INTEGER (SIZE 8 BYTES),
16    transmitTS       INTEGER (SIZE 8 BYTES)
17 } (ENCODED BY CUSTOM)
18 <transfer>
19    Back{ (flags & 56) == 32 }
20 </transfer>
21
22 END
```

Figure 2.1: Defining NTP version 4 in SCL.

```
1  UDP DEFINITIONS ::= BEGIN
2
3      IMPORTS PDU FROM RTPS, PDU FROM NTPV4;
4
5  PDU ::= (RTPS.PDU | NTPV4.PDU)
6
7  END
```

Figure 2.2: Unifying modules that run on top of the UDP protocol.

2 of the networking model, all the way up to application level data in layer 7. This gives the end user the flexibility of specifying which data they want to generate a parser for.

```
1  UDP DEFINITIONS ::= BEGIN
2
3      IMPORTS PDU FROM RTPS, PDU FROM NTP;
4
5  PDU ::= SEQUENCE {
6      srcPort     INTEGER (SIZE 2 BYTES),
7      dstPort     INTEGER (SIZE 2 BYTES),
8      length      INTEGER (SIZE 2 BYTES),
9      checksum    INTEGER (SIZE 2 BYTES),
10     udpProto    UDP_PROTO (SIZE DEFINED)
11 } (ENCODED BY CUSTOM)
12
13 UDP_PROTO ::= (RTPS.PDU | NTP.PDU)
14
15 END
```

Figure 2.3: Including UDP layer 4 parsing for the generator.

2.5.1 ASN.1 Keywords

Protocol specifications in ASN.1 use multiple keywords to help describe a module and define data types. The following sections will describe some of the keywords and types relevant to the protocol parser and the extended SCL language used. All keyword examples are shown in Figure 2.4.

Types

There exists type decisions where a choice needs to be made inside of a protocol specification in ASN.1. The protocol parser also uses four different types allowed by ASN.1. These are the `INTEGER`, `OCTET STRING`, `REAL`, and user-defined `SEQUENCE` object types.

SEQUENCE: A `SEQUENCE` is frequently used in the protocol parser, and allows the user to define their own type in ASN.1 and SCL. Sequences will hold a list of

```
1  RTPS DEFINITIONS ::= BEGIN
2
3      EXPORTS PDU;
4
5  PDU ::= (FULL | PING)
6
7  FULL ::= SEQUENCE {
8      Header      HEADER (SIZE DEFINED),
9      guidPrefix  GUIDPREFIX (SIZE DEFINED),
10     subMsg      SET OF SUBMESSAGE (SIZE CONSTRAINED)
11 }
12 <transfer>
13     Back { Header.protoName == 'RTPS' || Header.protoName == 'RTPX' }
14     Forward { END(subMsg) }
15     Callback
16 </transfer>
17
18 HEADER ::= SEQUENCE {
19     protoName   OCTET STRING  (SIZE 4 BYTES),
20     version     INTEGER (SIZE 2 BYTES),
21     vendorId    INTEGER (SIZE 2 BYTES)
22 }
23
24 SUBMESSAGE ::= (DATAPSUB | DATASUB | ACKNACK | HEARTBEAT |
25                 INFO_DST | INFO_TS | DATAWSUB | DATARSUB | GAP)
26                             .
27                             .
28                             .
29 END
```

Figure 2.4: Partial RTPS definition.

attributes that are either primitive types with known sizes, or other user- defined `SEQUENCE` types. The `FULL` sequence starts at line 7 for example. It holds all of the attributes and constraints until line 16.

Type Decision: A Type decision is a user-defined object that can be one of multiple user-defined `SEQUENCE` types. It is used to allow for multiple choices inside of a module for the protocol parser. Lines 5 and 24 show two different type decisions.

INTEGER: The `INTEGER` is a primitive type which specifies that an attribute is a numeric type. It is only used inside of user-defined `SEQUENCE` objects in the protocol parser. This is seen in line 20.

OCTET STRING: This is a primitive type which specifies that an attribute is raw data that is not an integer value. The protocol parser assumes `OCTET STRING` attributes are 8-bit text, where one character in the string is one byte. Other text encodings can be defined by custom `SEQUENCE` types for the protocol parser. Line 19 shows an `OCTET STRING` attribute of the `HEADER` sequence.

REAL: A `REAL` is used in cases where the data being parsed is floating-point data. The precision of the floating-point is determined by the number of bytes specified for the attribute.

SET OF: This is a prefix keyword to any of the primitive or user-defined types inside of a `SEQUENCE`. This indicates that there exists a list of the specified type for an attribute. This can be seen in line 10, where the `subMsg` attribute holds a list of the `SUBMESSAGE` user-defined type. In this case, `SUBMESSAGE` is a type decision, allowing the `subMsg` attribute to hold a list of any of the types in the type decision.

2.6 The Old SCL Language

This section describes the relevant parts of the old SCL language that are used in the parser generation framework.

2.6.1 Constraints

Constraint blocks are written as XML tags directly following sequences in SCL. The constraint type used for the parser generator, is the `transfer` block, line 12 in Figure 2.4. This block type as described by Marquis et al. and is used for constraints that specifically pertain to the encoding or decoding of data. The SCL language also introduces other types of constraint blocks which are not needed to generate the context-sensitive parsers. In this case, only `transfer` blocks are used to properly decode the data. All of the examples provided below are from Figure 2.5 unless otherwise specified.

Value Constraints

The parser can decide if it is decoding the correct type of data if it knows what values it should expect for certain fields. Value constraints will tell the parser to check for a value as soon as the specified field is parsed. This is seen in Figure 2.5 line 14. In this case, if the one byte integer field `kind` is not the value specified, then the parser knows that the data currently being parsed is not of the `DATASUB` type in RTPS, and the parser must backtrack. Value constraints can be specified for any attribute inside of a `SEQUENCE` type, and will cause the parser to immediately check the value after the given field is parsed.

Length Constraints

Length constraints exist when the length of a field depends on a previously parsed value. Figure 2.6 shows two sequences that both have length constraints. The `NESTEDSTRING` sequence is used as the type for the `topicName` field in the

```
1  DATARSUB ::= SEQUENCE {
2      kind       INTEGER (SIZE 1 BYTES),
3      flags      INTEGER (SIZE 1 BYTES),
4      nextHeader INTEGER (SIZE 2 BYTES),
5      extraFlags INTEGER (SIZE 2 BYTES),
6      qosOffset  INTEGER (SIZE 2 BYTES),
7      readerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
8      writerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
9      writerSEQ  INTEGER (SIZE 8 BYTES),
10     inlineQos  QOSPARM (SIZE DEFINED) OPTIONAL,
11     serializedData TOPICS (SIZE DEFINED) OPTIONAL
12 } (ENCODED BY CUSTOM)
13 <transfer>
14 Back {kind == 0x15}
15                     .
16                     .
17                     .
18 </transfer>
```

Figure 2.5: RTPS definition of DATARSUB sequence.

PIDTOPICNAME sequence. This constraint allows users to deal with dynamically sized data that may be decided at run time for these protocols. This constraint type also introduces the concept of nested length constraints which can be a serious security issue. We discuss the importance of our source code implementation of this issue in chapter 4.

Backward/Forward Constraints

Figure 2.6 lines 7 and 8 show the concept of forward and backward constraints. These were added to constraints inside of transfer blocks. Back constraints indicate that a constraint needs to be satisfied after parsing the attributes inside of the block. In the case of line 7, the ParameterKind attribute can be checked only after the attribute has been parsed. A forward constraint indicates that the constraint needs to be satisfied before an attribute is parsed. The attribute in question is

```
1  PIDTOPICNAME ::= SEQUENCE {
2      parameterKind    INTEGER (SIZE 2 BYTES),
3      parameterLength  INTEGER (SIZE 2 BYTES),
4      topicName        NESTEDSTRING (SIZE DEFINED)
5  }
6  <transfer>
7  Back {parameterKind == 5 }
8  Forward { LENGTH(topicName) == parameterLength }
9  </transfer>
10
11 NESTEDSTRING ::= SEQUENCE {
12     nameLength      INTEGER (SIZE 4 BYTES),
13     name            OCTET STRING (SIZE CONSTRAINED),
14 }
15 <transfer>
16     Forward { LENGTH(name) == nameLength }
17 </transfer>
```

Figure 2.6: Example of nested Length Constraints.

always the one that indicates what type of forward constraint it is. In the line 8 example, the length constraint needs to be checked just before `topicName` is parsed.

2.7 Previous Work on Protocol Parsing

There exists an older protocol parser created by Sylvain Marquis [23] that also uses SCL to define the protocol modules. This parser was a general SCL interpreter. The purpose of the general interpreter was to validate and to mutate network packets. The project provided a penetration testing framework for different networking protocols. It was mainly used for the offline testing of network data through packet captures, and was viable for its purpose. The old parser was first used in the beginning of this project to attempt to parse offline network data through SCL descriptions. while the functionality could be extended to support new protocols and constraint types in SCL, the requirements of this project could not be fully met

with the older framework. To describe some of the issues, a general understanding of the parsing method the framework uses needs to be established.

The old parser is written in C++ as a general SCL interpreter that reads SCL descriptions serialized in XML. It creates two trees, a grammar tree, and a concrete tree. The grammar tree is used to represent a protocol defined in SCL inside of an internal data structure, and allows the parser to understand what the data should look like for this protocol. It is used to create and validate the concrete tree, which holds the final data that is parsed from a packet. Figure 2.7 shows the grammar objects that are used as nodes in the grammar tree. A tree is built based on SCL, where each node represents a type in SCL. Type decisions or choices are represented as `GrmOr` nodes. Sequences in SCL are represented using `GrmRec` nodes. The `GrmToken` object represents all of the sequence attributes in SCL that will hold the actual binary data from a packet.

The grammar tree is created by deserializing the XML file for the protocol, and iterating through all of the different SCL types. Once attributes inside of SCL sequences are reached, nodes are allocated as `GrmToken` types, and the size and type of the attributes are stored in the grammar node. All `GrmToken` types are stored as children inside of `GrmRec` nodes which represent sequences. By iterating and deserializing the XML file that describes the protocol, the internal grammar tree can be used to create the concrete tree and consume bytes from a network packet.

Figure 2.8 shows the base object for a node in the concrete tree. The child type of the concrete object is determined by looking at the SCL grammar for a specific protocol. A `ConReq`, or a concrete record, represents a sequence in SCL. A `ConToken`, or concrete token, represents a token that holds binary data parsed from the PDU.

```
1   class Grm {
2      public:
3         const char * user_type;
4         virtual Grm() = 0;
5   };
6
7   class GrmToken: public Grm {
8      public:
9       const char * type_name;
10      int byte_size;
11      int bit_size;
12      int term;
13      GrmTextRep  text_rep;
14      int encoding;
15  };
16
17  class GrmContainer: public Grm {
18     public:
19        struct Field{
20            const char    * name;
21            unsigned long nameIndex;
22            Grm * child;
23            Field(){
24               name = NULL;
25               child = NULL;
26               nameIndex = getNullIndex();
27        }
28        Field(const char * n, Grm * c){
29            name = n;
30            nameIndex = hashForString(n);
31            child = c;
32        }
33        Field setVals(const char * n, Grm * c){
34            name = n;
35            nameIndex = hashForString(n);
36            child = c;
37            return *this;
38        }
39     };
40     std::vector<Field> members;
41     GrmContainer();
42  };
43
44  class GrmOr: public GrmContainer {};
45
46  class GrmRec: public GrmContainer {
47      public:
48        GrmRecType            rec_type;
49        std::vector<exprTree> back_constraints;
50        std::vector<exprTree> forward_constraints;
51  };
```

Figure 2.7: Some of the objects used to create the grammar tree.

```
1   // concrete hierarchy root
2   // contains any fields common to all nodes
3   class Concrete {
4       public:
5       const char *  name;
6       unsigned long nameIndex;
7       const char *  type;
8       unsigned int  size;
9       unsigned int  offset;       // position in the PDU
10      unsigned int  bit_offset;   // position in the PDU
11
12      public:
13          virtual Concrete() = 0;
14          virtual void walk(int,FILE *) = 0;
15
16      protected:
17          static const int  numspaces;
18          static const char *spaces;
19          static void indent(int level,FILE * file);
20  };
21
22  // ConToken class - represents the leaves of the
23  // concrete syntax tree
24  class ConToken: public Concrete {
25      public:
26        unsigned int  len;
27        unsigned int  bitlen;
28        unsigned long int_value;
29        bool          int_value_valid;
30        char *        str_value;
31
32        void walk(int l,FILE * f) {. . .};
33        ConToken();
34  };
35
36  // ConRec - abstract record class only used as common
37  //  parent for Sequence and Set classes
38  class ConRec: public Concrete {
39      public:
40        std::vector<Concrete*>  members;
41        void walk(int l,FILE *f) {. . .};
42        ConRec(){. . .}
43  };
44
45  // ConSeq - use for SEQUENCE { ... }
46  class ConSeq: public ConRec{
47      public:
48        void walk(int l,FILE *f) {. . .};
49  };
```

Figure 2.8: Some of the Objects used to create the concrete tree.

`ConToken` instances are the leaf nodes in the tree that will hold the parsed data. The concrete tree structure recursively creates child nodes based on the types defined in SCL, and fills them in with data from the PDU in an attempt to properly consume the data according to the specification. The recursive function determines the next type defined in the description and edits the overall tree structure of the packet accordingly. This process repeats until all of the tree structure has properly been filled in without failing. If the leaf nodes fail to parse some data from the PDU, the parse of the whole packet fails and returns. For example, parsing a sequence in SCL would create a `ConSeq` object which would recursively call the `doToken` function that would create a `ConToken` node for each of the attributes inside of the SCL sequence. The grammar tree is referenced to check this specific sequence for the attributes that exist and how many bytes should be consumed for each.

For the framework to run on live network data, the general interpreter would have to read SCL grammars every time a packet arrives depending on the packet's protocol, and would create and walk a tree of objects as defined in the SCL description for that protocol.

The approach of writing one universal SCL interpreter and using XML libraries to deserialize SCL specifications could not provide the needed performance for a live network protocol parser to be used in an NIDS system with large amounts of traffic. Through some testing, it was determined that the performance of the general interpreter could be improved by avoiding a significant number of string comparisons in the algorithm. An efficient hash function was used to initialize all string data and to use a hash table to refer to the strings, allowing for O(1) string comparisons. while this increased the performance of the interpreter to support

a 50 Mbit connection, the requirement of creating a network protocol parser that would support gigabit speeds would be unattainable with the general interpreter used by Marquis et al.. This is the main motivation behind creating the parser generation framework.

2.8 Related Work

There has been extensive work in the domain of parser generation using domain-specific languages including Nail [2], ANTLR [27], Autumn [20], Spicy [31], or Hammer [34].

2.8.1 Nail

Nail is a parser generator that focuses on language safety. The generator introduces similar constraint types that deal with context-dependent fields and field lengths. There exist some differences in our approach that include an LL(1) optimization before parser generation, and the ability for custom callback functions that provide an extensible framework. We also ensure that nested length constraints cannot violate parent PDU lengths to help mitigate possible attack vectors through the generated parsers.

2.8.2 ANTLR

While general-purpose parsing tools like ANTLR provide a viable framework to generate LL(k) recursive descent parsers, we have provided a modular solution that is specialized for binary protocols and allows users to write custom callback functions to deal with different parse types. ANTLR may very well be used to parse

the same packet types and is more generalized. In our case we focus on providing a specification language that is familiar to network protocol developers and users. This allows network engineers to understand the extended SCL language with ease, and avoids them having to learn a completely new domain specific language. The other main difference is that in our system, the input language is to be used for the full NIDS system, and not only the parser. While ANTLR can provide the interface to the network parser, another input language would have to be used to generate code for the constraint engine in our network architecture. By reducing the reliance on other software systems, library dependencies are minimized, and a multi-platform solution can be achieved.

2.8.3 Autumn

Autumn is a context-sensitive parsing library that supports backtracking and memoization. It defines six primitive opterations that the parsers can carry out using the library's API. These are call, snapshot, diff, transform, restore, and merge. We briefly describe each of these operations. The call operation represents the invocation of a new parser in the system which in the context of network protocol parsing can be the invocation of a parser for the next layer in the networking model. The snapshot operation will capture the state of the parse at a specific stage during the execution of the parser. The diff operation will return an object that represents the difference between a given snapshot and the current state of the parser. Differences in states, or delta's as defined by Laurent et al. can later be used in other operations. The transform operation is used to allow parsers to change state arbitrarily. The

restore operation allows the state of the parse to change based on the input snapshot provided. The merge operation finally takes a delta object that is returned by the diff operation as input and returns the aggregation of the delta and the current state. Autumn provides an API that is feasible in creating context-sensitive parsers. It's internal representation of the data is still manually created at the moment.

2.8.4 Spicy

Spicy is a very recently published DPI framework that aims to also generate parsing code from a specification language. There are multiple similarities and differences in both of our approaches that we point out. Similar to our work, Spicy also ensures robust error handling on parse failure, and allows integration with other software through a custom callback mechanism. They also introduce similar parsing constraints that allow for LL(1) optimizations to be made. Protocols can also be layered as they are modular in Spicy's domain specific language. Spicy allows for the dynamic detection and dissection of protocols through their concept of sinks. They provide support through hooks to generate debugging and support TCP stream reassembly.

SCL has a declarative constraint style that is focused more on a high-level extension to the industry standard ASN.1 notation. Sommer et al. [31] seem to suggest that there is some manual intervention in backtracking (see `&try` attribute notation).

We do not allow the user to explicitly state how values are converted, this is decided by the parser generator and the type provided in ASN.1. Finally we allow for the specification of choices between types based on sub-elements of complex

user-defined types. This can be specified adjacent to the type that is being selected inside of sequences. These nested types of value constraints are discussed in further detail in the next two chapters.

Code Generation

Spicy's code generation is achieved by using the HILTI compiler tool chain [32] to generate native code. HILTI is an abstract machine model used to aid in the development of traffic analysis tools. It is built on top of the LLVM [11] tool chain and translates programs written for HILTI into optimized C code. It supports just-in-time compilation in addition to static compilation. HILTI was created as a means to remove the requirement of low-level implementations for traffic analysis applications while ensuring that programs are secure, thread-safe, and high-performance.

The Spicy library itself is developed in C and C++ and is mainly responsible for translating spicy modules into HILTI code. The Spicy parser generation framework by itself consists of approximately 32,500 lines of code. It utilizes HILTI's memory management and concurrency model to define the Spicy parser. It works similarly to our research group's previously implemented protocol parser by reading a protocol's grammar into an internal data structure. The grammar data structure however is then used to generate incremental parsing code in the HILTI engine in the form of HILTI op codes rather than creating an abstract syntax tree. The end result becomes native C code generated by HILTI that closely resembles hand written network parsing code. Data packets are consumed by using sliding window buffers, and tokens are parsed incrementally. The sliding window is provided to allow data to remain available for back reference. Iterators are used to consume the

buffers and buffer memory is deallocated once bytes are processed. Data can only be accessed inside of the current starting and ending iterators in this case. Length constraints are processed by creating nested iterators that will limit the number of bytes to the length specified. This avoids the possibility of overflowing lengths by spoofing nested length values to be greater than parent length constraints.

Our parser generator's approach is to keep a full buffer of the current packet in memory until the full packet is parsed into internal data structures. In our model, parsing can only occur incrementally, and previous data can only be referenced if they have been successfully tokenized and parsed. Our parser will deal with nested length constraints by similarly limiting access to the packet's PDU based on the length constraint. This also avoids overflowing possibilities. This is discussed in more detail in chapter 4.

Spicy does not employ any recursive backtracking in the case of failed parses. The Spicy generator will attempt to create an LL(1) look-ahead parser by evaluating the first field of different types when given a choice between multiple types. This requires that the user provide value constraints on each of the fields in order for Spicy to properly create the look-ahead parser. Otherwise, the engine requires that the user manually specifies where the parser should backtrack to before attempting a different parsing choice. If there doesn't exist enough information to generate a look-ahead parser and there is no manual backtracking written into a specification, a parse will generate an error if it fails.

Spicy provides a viable framework to generate protocol parsers just in time using the HILTI engine. While both the final outputs between HILTI's generated native parser and our generated parser are similar to hand written parsers, our

main difference is the the input specification language. SCL does not require knowledge of low level programming types. Our approach to translating the input specification language relies on source transformation in TXL as compared to code generation into an intermediate language for an abstract machine.

2.8.5 Others

Hammer is also an example of a library that allows for the creation of LL(k) recursive descent parsers. It is an open source project specifically created for the parsing of binary protocols, the main difference being that its input specification is written directly in C.

We have introduced the idea of nested endianness, where the byte-order of fields can change based on previously parsed data. This allows for the ability to generate parsers for complicated binary protocols like RTPS for DDS, and DRDA [15]. While there exist multiple tools that can achieve the requirements for the parser to be secure, fast, and reliable, our solution suggests an expressive high-level specification language that closely relates to network protocol specification languages like ASN.1. Our main contribution to SCL is the ability to use context-dependent constraints to generate the parsers. This makes protocol specifications simple to implement. We have extended SCL to also allow for inter-packet constraints, which will generate IDS rules that can reference data between different packets. This allows the total function of our NIDS system to be specified using one input format, SCL, and removes the need for multiple dependencies in creating a specialized NIDS for limited networks.

Chapter 3

Syntax Constraint Language

This chapter describes the newly extended version of the SCL language that has been created as the specification language for this project. These new additions are essential for the proper generation of the protocol parsers. We provide a general overview of the new additions, and the shortcomings of the original SCL language. The desired characteristics of the whole project's input and output are also discussed. This chapter acts as a precursor for the chapter that follows on a detailed description of the source translation pipeline. The goal in this chapter is to describe the semantic meaning of the newly added definitions. These definitions are essential in generating optimized parsers.

3.1 SCL Additions

In this section we describe the main extensions to SCL that are used to provide the information needed to generate the parser. We first discuss extensions that are made directly to the ASN.1 subset of SCL, followed by the extensions to the SCL constraint blocks.

A type decision is a grammar production with multiple choices, and will be referred to as a type decision for the rest of the thesis. A sequence is a user-defined type in SCL that describes all of the fields present in the data type, their own type and their size. Constraint blocks can be appended directly to type decisions and sequences, lines 5 and 7 in Figure 3.1, respectively.

3.1.1 Custom Extensions

There are three extensions made to SCL in order to properly generate the parsers. We describe these below.

Size Constraints

The first involves specifying SCL size constraints directly following attribute types in ASN.1 notation as shown in Figure 3.1 line 8. This avoids having to take the approach introduced by Marquis et al. [21] of specifying the number of bytes of each attribute in an XML block following a sequence.

Force Byte-Order

The second extension specifically forces a certain byte order for the parsing of an attribute. The parser generator provides the capability of specifying nested endianness inside protocols. This can be seen in protocols like RTPS, which may change their byte ordering based on values inside of the data that is being parsed. Nested endianness and their constraints will be described in the next section.

```
1  IGMP DEFINITIONS ::= BEGIN
2
3     EXPORTS PDU;
4
5  PDU ::= ( Query | V2Report | V2Leave | V3Report)
6
7  Query ::= SEQUENCE {
8     type         INTEGER (SIZE 1 BYTES),
9     maxRespTime  INTEGER (SIZE 1 BYTES),
10    checksum     INTEGER (SIZE 2 BYTES),
11    groupAddr    INTEGER (SIZE 4 BYTES),
12    v3Add        V3Addition (SIZE DEFINED) OPTIONAL
13 } (ENCODED BY CUSTOM)
14 <transfer>
15    Back{type == 17}
16    Forward { EXISTS(v3Add) == PDUREMAINING }
17 </transfer>
18
19 V3Addition ::= SEQUENCE {
20    resvSQRV     INTEGER (SIZE 1 BYTES),
21    QQIC         INTEGER (SIZE 1 BYTES),
22    numSources   INTEGER (SIZE 2 BYTES),
23    srcAddrs     SET OF SOURCEADDRESS (SIZE CONSTRAINED)
24 } (ENCODED BY CUSTOM)
25 <transfer>
26    Forward{ CARDINALITY(srcAddrs)  == numSources }
27 </transfer>
28
29 SOURCEADDRESS ::= SEQUENCE {
30   srcAddr       INTEGER (SIZE 4 BYTES)
31 } (ENCODED BY CUSTOM)
32
33 END
```

Figure 3.1: Partial IGMP definition with SCL constraints.

Optional Fields

Some protocols have attributes which only exist in some cases. These can be additions to the end of a packet that will not always appear. They can also be fields that only exist when previous fields hold specific values. The IGMP module in Figure 3.1 shows the example where the v3Add attribute is an optional field. This

field only exists in a query packet of the IGMPv3 protocol, and does not exist in the previous IGMPv2 version. By stating the attribute as `OPTIONAL` in line 12 of Figure 3.1, the parser generator knows that this field will only exist when a specific constraint is met. This leads to checking the constraints specifically in the `Query` sequence of the IGMP protocol. The `OPTIONAL` keyword is actually part of the original ASN.1 specification, and has been implemented in this version of SCL to generate code for fields that do not always exist in the data.

3.1.2 Constraints

As described in the previous chapter, the `transfer` constraint blocks are used in order to specify constraints on attributes in sequences and type decisions in SCL. Due to the addition of attribute sizes directly to SCL, the `size` XML block used by Marquis et al. [21] is no longer needed for the new version of SCL. The constraints specified below are part of the new contribution to the SCL language, and are essential to the generation framework's function.

Terminators

Figure 3.2 shows the example of fields that can have a variable number of objects. In the case of the topicData attribute in line 4 of Figure 3.2, there can be multiple fields of the TOPICPARMS parent-type defined by the user. The SET OF prefix in SCL is used to define a list of multiple fields of the same type. In order to parse `SET OF` fields properly, the generated parsers require that a terminating constraint is specified in the transfer block. The example at line 7 of Figure 3.2 says that the last user-defined type that should be in the list of topicData is the PIDSENTINAL

```
1  TOPICS ::= SEQUENCE {
2      encapsKind INTEGER (SIZE 2 BYTES) BIGENDIAN,
3      encapsOpts INTEGER (SIZE 2 BYTES) BIGENDIAN,
4      topicData SET OF TOPICPARMS (SIZE CONSTRAINED)
5  } (ENCODED BY CUSTOM)
6  <transfer>
7  Forward { TERMINATE(topicData) == PIDSENTINAL}
8  </transfer>
```

Figure 3.2: RTPS definition of TOPICS sequence.

```
1  TOPICPARMS  ::= ( PIDTOPICNAME | PIDTYPENAME |
2                  PIDRELIABILITY | PIDENDPOINTGUID |
3                  . . . |PIDSENTINAL )
```

Figure 3.3: partial listing of the TOPICPARMS type decision.

type, a child of the TOPICPARMS type as seen in Figure 3.3. There are two other types of terminator constraints that may be used to specify when a SET OF parse should end. The first is a CARDINALITY constraint, where the number of items in a list are specified in a previously parsed field. This can be seen in line 26 of Figure 3.1, the number of items in the srcAddrs field depends on the value parsed in the numSources field. The last possibility for a SET OF field is that the field with a list of data is at the very end of a packet. This is specified as follows:

```
FORWARD{ END( field ) }
```

The CARDINALITY constraint was the only type of terminating constraint that previously existed in Marquis' work [21].

Nested Constraints

A nested constraint is a value or length constraint where the value that needs to be checked is inside of a sub-type. In the case of line 15 in Figure 3.4, the `key` data

```
1  DATARSUB ::= SEQUENCE {
2      kind       INTEGER (SIZE 1 BYTES),
3      flags      INTEGER (SIZE 1 BYTES),
4      nextHeader INTEGER (SIZE 2 BYTES),
5      extraFlags INTEGER (SIZE 2 BYTES),
6      qosOffset  INTEGER (SIZE 2 BYTES),
7      readerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
8      writerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
9      writerSEQ  INTEGER (SIZE 8 BYTES),
10     inlineQos  QOSPARM (SIZE DEFINED) OPTIONAL,
11     serializedData TOPICS (SIZE DEFINED) OPTIONAL
12 } (ENCODED BY CUSTOM)
13 <transfer>
14 Back {kind == 0x15}
15 Back {writerEnt.key == 0x4 }
16 Back {writerEnt.kind == 0xC2 }
17 Forward { ENDIANNESS == flags & 0x1 }
18 Forward { EXISTS(inlineQos) == flags & 0x2 }
19 Forward { EXISTS(serializedData) == flags & 0xC }
20 </transfer>
```

Figure 3.4: RTPS definition of DATARSUB sequence.

is inside of the ENTITYID type that is parsed for the writerEnt field. The value of a nested constraint is checked once the whole field at the current level has been parsed, or once writerEnt is completely parsed in this case.

Endianness

Figure 3.4 provides the example of nesting the endianness of data in the PDU. Line 17 in Figure 3.4 shows that the endianness of the attributes following the `flags` attribute are based on the value that is parsed for the attribute itself. We provide the capability of ignoring the current byte-order to parse in by specifying that an attribute must be parsed using a specific byte-order as seen in line 7.

Callbacks

When packets have been parsed, users are given the choice of specifying callback constraints in `transfer` blocks, allowing the parser to call a function that will process the parsed packet. By providing callback functions, users of our generated parsers do not have to walk our data structures in order to determine what type of network data has been dissected, and can directly proceed with processing this type of packet.

3.2 Previous SCL Shortcomings

The additions made to the old SCL language not only provide the information needed to generate the parsers, they also reduce the lines of translation code needed to generate the parsers. Additions like moving the size of the fields beside their types, and containing all of the new constraints inside one type of XML block reduce the need of searching for the required information in multiple places in the specification. This makes the translation code shorter and more efficient.

Most of the shortcomings of the original SCL language pertain to the constraint types that are needed to properly generate context-sensitive parsers. This includes optional fields coupled with `EXISTS` constraints, nested endianness, endianness and terminating constraints, and the need to specify which data types will be provide the callback interfaces for the parsed data. This project has taken a context-sensitive grammar in SCL and added support for the generation of context-sensitive parsers.

The method used to allow modularity in the previous version of SCL was not fully tested. All SCL specifications were compiled into one SCL specification, and imports/exports were not fully validated. By Keeping all specifications and

modules in their own files, we are still able to provide full modularity during the analysis, and properly validate imports and exports. The new translation and analysis engine also allows for easier maintenance since all modules are analysed separately. Interfaces are generated between modules seamlessly. The current application of out modularity has been used in the UDP module shown in this chapter. While the new grammar additions and modularity support provide essential information for the generation of context-sensitive parsers, there are still improvements that need to be made to the semantic phase of the analysis. This is discussed in future work.

3.3 Generation Framework

We introduce the process of the generation framework at a high level in this chapter, and provide in depth descriptions of the full pipeline in the next chapter. The process begins with all of the network protocol descriptions in the new SCL language provided to the user. Protocols can be layered accordingly depending on which layers of the network protocols the user wishes to parse. The list of module specifications are fed into the analysis pipeline separately to generate a set of separate header and source files in C that interface together according to the user's specification. We describe the process for a single module, and continue by depicting how all modules interface together in the source code. We also describe the desired output for the callback interface that allows the user to interact with the data that is saved by the parser.

3.3.1 Unique Naming

The process begins by taking a single module as input and annotating the file accordingly. The annotations ensure that a module is uniquely named and avoids any name collisions inside of a module and between modules. This ensures that any types or fields holding identical names inside of modules do not collide and can be differentiated properly in the specification analysis and code generation. The unique naming process proceeds by renaming all declarations inside of a module first, followed by the renaming of all references.

3.3.2 Data Structure Generation

Following unique naming annotations in the SCL modules, the information required to generate a header file for the module is present. The header file holds all of the data structure definitions needed to save the information from a parsed protocol. The data structures generated are then used when parsing the data, and when the user need to access the data following the parsing phase. All exported types will include their parsing function's declaration inside of the header file for other modules to be able to use. Figure 3.5 shows the data structure generation of the `DATARSUB_RTPS` sequence type.

3.3.3 LL(1) Optimization

A second phase of annotations analyses the module for optimizations that can be made to create a more efficient parser. This optimization phase attempts to ensure that a look-ahead parser can be generated from the information present in the module specification. The analysis will copy needed information to certain parts

```
1  typedef struct {
2      uint8_t kind;
3      uint8_t flags;
4      uint16_t nextheader;
5      uint16_t extraflags;
6      uint16_t qosoffset;
7      ENTITYID_RTPS readerent;
8      ENTITYID_RTPS writerent;
9      uint64_t writerseq;
10     QOSPARM_RTPS *inlineqos;
11     TOPICS_RTPS *serializeddata;
12 } DATARSUB_RTPS;
```

Figure 3.5: C struct translation of example in Figure 3.4.

of the specification to make the translation phase easier. This helps separate the optimization analysis from the parser generation itself, which allows the overall project to be maintainable.

3.3.4 Parser Generation

Once the optimization analysis is complete, a source file is generated to parse the actual data for this module. All SCL modules are translated in place in the order that the specification is written. Only exported types can be directly called from other protocol parsers. This ensures that the intended modularity written by the used holds during parser generation. This is described below.

Interface Between Modules

Interfacing between modules depends on the types that are exported and imported in between modules as specified by the user in the SCL specifications. A parser will be able to call the respective parse functions of all imported types inside of a module. A module that has exported certain types will provide the functions

```
1  #define BIGENDIAN (0x0)
2  #define LITTLEENDIAN (0x1)
3  #include "UDP_Generated.h"
4  #include "pglobals.h"
5  #include "putilities.h"
6  bool parseUDP (PDU_UDP *pdu_udp, PDU *thePDU, char *progname, uint8_t endianness);
7  void freePDU_UDP (PDU_UDP *mainpdu);
8
9  bool parseUDP (PDU_UDP *pdu_udp, PDU *thePDU, char *progname, uint8_t endianness) {
10     unsigned long pos = thePDU->curPos;
11     unsigned long remaining = thePDU->remaining;
12     if (parseRTPS (&pdu_udp->ptr.pdu_rtps, thePDU, progname, endianness)) {
13         pdu_udp->type = PDU_RTPS_VAL;
14         return true;
15     }
16     thePDU->curPos = pos;
17     thePDU->remaining = remaining;
18     if (parseNTP (&pdu_udp->ptr.pdu_ntp, thePDU, progname, endianness)) {
19         pdu_udp->type = PDU_NTP_VAL;
20         return true;
21     }
22     return false;
23 }
24
25 void freePDU_UDP (PDU_UDP *mainpdu) {
26     if (mainpdu->type == PDU_RTPS_VAL)
27         freePDU_RTPS (&mainpdu->ptr.pdu_rtps);
28     else if (mainpdu->type == PDU_NTP_VAL)
29         freePDU_NTP (&mainpdu->ptr.pdu_ntp);
30 }
```

Figure 3.6: UDP generated parser according to imports in SCL.

that are exported inside of its header files, allowing for other parsers to call these functions. Figure 3.6 shows the generated parser for the UDP protocol as a result of the UDP SCL specification shown in the previous chapter. As seen on lines 12 and 18, the imported module types PDU from NTP and RTPS can directly be called from the UDP parser.

In the case of the current translation process, it is assumed that each module will only export one of its types, usually its top level type. For this reason the

translated function interface for imports assumes the imported modules rather than the imported type name. In this example it is `parseRTPS` and `parseNTP` rather than `parsePDU_RTPS` and `parsePDU_NTP`. This is a shortcoming of the current translation framework. The SCL analysis phase fully supports multiple exports for modules, and the translation engine will need to ensure that this is possible in the source code generation as part of future work. This way unique types that are declared in one module can be used in multiple other modules without having to be redefined. For the purposes of the protocols this project has supported up to this point, none of the protocols have had identical types defined.

3.3.5 Callback Interface

An important aspect of the parser generator framework is how the generated parsers provide access to parsed packets to allow the integration of the parsers to other systems. Given that the generated parsers provide data at the packet level with high performance, it is essential that the design to the callback interface avoids doing extra work. The framework will generate a callback function call at the end of a packet parse. By allowing each of the different protocols to create their own callback function, the user does not have to walk the data structures of a parsed packet to figure out what the packet type is. The callback interface avoids the need to duplicate the work that is already done by the parser. Each callback function can be specialized for each packet type by the user. Figure 3.7 shows an example of the IGMP callback functions being called once the packets are parsed. Figure 3.8 shows the actual callback interface which is used by the NIDS constraint engine to check the data for malicious content. The callback interface is one file that groups all of

```
1  switch (type) {
2  case 34 :
3      if (parseV3Report_0 (&pdu_igmp->ptr.v3report_igmp, thePDU, progname, &type, endianness)) {
4          pdu_igmp->type = V3Report_IGMP_VAL;
5          V3Report_IGMP_callback (&pdu_igmp->ptr.v3report_igmp, thePDU);
6      } else {
7          return false;
8      }
9      break;
10 case 17 :
11     if (parseQuery_0 (&pdu_igmp->ptr.query_igmp, thePDU, progname, &type, endianness)) {
12         pdu_igmp->type = Query_IGMP_VAL;
13         Query_IGMP_callback (&pdu_igmp->ptr.query_igmp, thePDU);
14     } else {
15         return false;
16     }
17     break;
18 case 22 :
19     if (parseV2Report_0 (&pdu_igmp->ptr.v2report_igmp, thePDU, progname, &type, endianness)) {
20         pdu_igmp->type = V2Report_IGMP_VAL;
21         V2Report_IGMP_callback (&pdu_igmp->ptr.v2report_igmp, thePDU);
22     } else {
23         return false;
24     }
25     break;
26 case 23 :
27     if (parseV2Leave_0 (&pdu_igmp->ptr.v2leave_igmp, thePDU, progname, &type, endianness)) {
28         pdu_igmp->type = V2Leave_IGMP_VAL;
29         V2Leave_IGMP_callback (&pdu_igmp->ptr.v2leave_igmp, thePDU);
30     } else {
31         return false;
32     }
33     break;
34 default :
35     return false;
36 }
```

Figure 3.7: UDP generated parser according to imports in SCL.

the protocols that the user has specified, and is the main output interface between the parser and any other networking application that is using the DPI framework.

```
1  void V3Report_IGMP_callback(V3Report_IGMP * v, PDU * thePDU) {
2    //Process a V3Report packet
3  }
4
5  void Query_IGMP_callback(Query_IGMP * q, PDU * thePDU){
6    //Process a Query packet
7  }
8
9  void V2Report_IGMP_callback(V2Report_IGMP * v, PDU * thePDU) {
10   //Process a V2Report packet
11 }
12
13 void V2Leave_IGMP_callback(V2Leave_IGMP * v, PDU * thePDU) {
14   //Process a V2Leave packet
15 }
```

Figure 3.8: IGMP callback interface.

Chapter 4

Source Translation

This chapter provides an in depth view of the full translation pipeline from the initial SCL description to the generated code output. The final output for each protocol is two files. The first is a header file that defines all the C structures used to represent parsed data. The second is a source file that parses data from the PDU. If the network data provided fails to parse, then the parser returns false. Flagged packets can be used to fix mistakes in the SCL specification, or to ensure that packets are malformed when parses fail. The parsed packets can then be used for analyzing the network data. In the scope of this project, the generated parsers will provide the data needed for the constraint-based IDS engine that is used to generate network alerts. The full pipeline can be seen in Figure 4.1

In this chapter, each stage in the pipeline will be described as follows. A code sample of the input will be shown with a description of what happens at the current stage. This is followed by an example of the output, and samples of TXL code that shows how certain parts of the source transformation for this stage were achieved. TXL samples are shown in sections that use different techniques to attain the desired output.

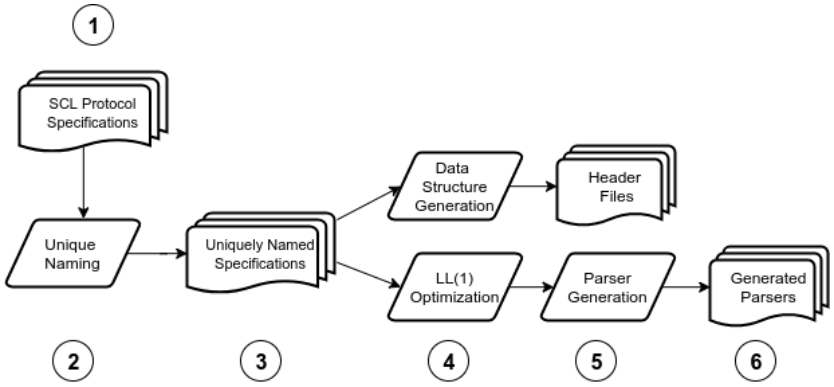

Figure 4.1: Parser generation chart.

4.1 Unique Naming

When protocol specifications are being written, it is possible that there exist duplicate names inside of modules and between different modules. It then becomes important to be able to differentiate declarations and references inside of modules when they have identical names. It is possible that the same name might be used for a field in more than one protocol. A similar scheme is implemented in programming language compilers. During compilation, a global variable should not collide with a local variable that has the same name and type in C for example. In order to ensure that the generated parsers do not have any name collisions between data types and modules specified in SCL, we need a unique naming scheme. Before SCL code is used for code generation, it is processed by a set of TXL scripts that will create unique names for all custom types defined inside of the SCL modules. This includes the module name, the sequence names, user-defined types, and field references in the constraint blocks. Dean et al. [6] employ a similar approach for unique naming when using TXL for source transformation.

There are three stages involved in the unique naming analysis of the SCL descriptions. The first deals with renaming all type declarations in SCL. The second involves renaming all of the references of renamed declarations inside of the SCL description. The last stage deals with the exporting of user-defined types inside of a module. Exported types are types that can be imported and used in other SCL modules, and are essential in allowing modules to be layered in SCL. The last stage ensures that modules can be used in other specifications by creating a table of exports for the current module, with the unique naming scheme. This last stage aids in the overall process of ensuring that `IMPORTS` can be referenced from other modules. this can be seen in Figures 4.2 and 4.3 where the `UDP` module imports the `PDU` user-defined type from the `RTPS` module in SCL.

Our renaming scheme is layered starting at the top level being the module name. We use underscores to indicate a level, and we add all of the levels to the end of a declaration or a reference. Levels are added from the bottom up all the way to the top level. The levels start with the SCL module definition name, followed by the sequence, or the type decision names. We keep a copy of old names beside a unique name separated by the caret ^ character. The original names are also used in the overall process to generate the parsers. Figure 4.3 shows the `FULL` sequence defined in the `RTPS` Module in SCL. We take the `Header` field in line 8 inside of the sequence as an example to be renamed. The final name moves up the levels starting with the sequence name, `FULL`, and up to the module name, `RTPS`. The unique name of the `type` field is `Header_FULL_RTPS`. We explore the details of all stages of the unique naming in the following subsections, and use snippets of the `RTPS` module in SCL to show the process. We also provide examples of the TXL

```
1                   UDP DEFINITIONS ::= BEGIN
2
3                       IMPORTS PDU FROM RTPS,  PDU FROM NTP;
4
5                   PDU ::= (RTPS.PDU | NTP.PDU)
6
7                   END
```

Figure 4.2: Partial UDP definition where other modules are imported.

```
1  RTPS DEFINITIONS ::= BEGIN
2
3      EXPORTS PDU;
4
5  PDU ::= (FULL | PING)
6
7  FULL ::= SEQUENCE {
8      Header      HEADER (SIZE DEFINED),
9      guidPrefix  GUIDPREFIX (SIZE DEFINED),
10     subMsg      SET OF SUBMESSAGE (SIZE CONSTRAINED)
11 } (ENCODED BY CUSTOM)
12 <transfer>
13     Back { Header.protoName == 'RTPS' || Header.protoName == 'RTPX' }
14     Forward { END(subMsg) }
15     Callback
16 </transfer>
```

Figure 4.3: Partial RTPS definition.

transformation code to generate the uniquely named output.

4.1.1 Declarations

The declarations stage of the unique naming involves all of the types in the ASN.1 notation that have been declared. This consists of type decisions, sequences, and sequence fields. In Figure 4.4, we have highlighted all of the declarations of the RTPS sample in SCL. These are the keywords that will be renamed at this stage. The output of the declaration renaming can be seen in Figure 4.5. As described,

```
1  RTPS DEFINITIONS ::= BEGIN
2
3      EXPORTS PDU;
4
5  PDU ::= (FULL | PING)
6
7  FULL ::= SEQUENCE {
8      Header      HEADER (SIZE DEFINED),
9      guidPrefix  GUIDPREFIX (SIZE DEFINED),
10     subMsg      SET OF SUBMESSAGE (SIZE CONSTRAINED)
11 } (ENCODED BY CUSTOM)
12 <transfer>
13     Back { Header.protoName == 'RTPS' || Header.protoName == 'RTPX' }
14     Forward { END(subMsg) }
15     Callback
16 </transfer>
```

Figure 4.4: Declarations in RTPS sample.

```
1  RTPS DEFINITIONS ::= BEGIN
2
3      EXPORTS PDU;
4
5  PDU_RTPS ^ PDU ::= (FULL | PING)
6
7  FULL_RTPS ^ FULL ::= SEQUENCE {
8      Header_FULL_RTPS ^ Header           HEADER (SIZE DEFINED),
9      guidPrefix_FULL_RTPS ^ guidPrefix   GUIDPREFIX (SIZE DEFINED),
10     subMsg_FULL_RTPS ^ subMsg           SET OF SUBMESSAGE (SIZE CONSTRAINED)
11 } (ENCODED BY CUSTOM)
12 <transfer>
13     Back {Header.protoName == 'RTPS' || Header.protoName == 'RTPX'}
14     Forward {END (subMsg)}
15     Callback
16 </transfer>
```

Figure 4.5: Renamed declarations in RTPS sample.

the fields inside of a sequence also include the sequence name before the module names.

Before the process of renaming declarations begins, we must check the SCL description for any underscores used in the naming of types and fields. Due to

underscores being used for the unique naming scheme, we substitute any existing underscores the user has provided in the description. Figure 4.6 shows the rule that will go through a full SCL module to replace underscores with the `'$'` character, a non existent character in SCL that is legal to use. We begin by searching all `[id]` types in SCL for the underscore character. Each occurrence is replaced, and an warning message is given to the user during the translation phase. This ensures that unique naming is not prone to error.

We proceed by creating separate renaming rules for each declaration type in SCL, which are fields, sequences, and type decisions. The example in Figure 4.7 shows how the module name of the SCL description is appended to the existing sequence name. In the replacement part of the rule, we keep the previous short name alongside the new one. There is also a call to the `renameType` rule, which will proceed with renaming all of the fields inside of a sequence. The new sequence name is given to the rule being called to be used as the new name to be appended to each field.

4.1.2 References

Once the first stage of unique naming is done, the renaming of the references begins. We continue with the RTPS example in Figure 4.5 as the input to the reference renaming stage. The remaining fields to be renamed are any usage of a user defined type inside of type decisions, sequences, and an analysis of the `transfer` constraint blocks. We also rename all imports and exports in place. In the case of the example used in the declarations phase, all of the references to the sequence types inside of the `PDU_RTPS` type decision, the reference to all of the types

```
1  rule noUnderscores
2      replace $ [id]
3          ID [id]
4      construct US [number]
5          _ [index ID "_"]
6      deconstruct not US
7          0
8      construct Err [id]
9      _ [+ "Warning, \""] [+ US] [+ "\" declaration \""]
10         [+ ID] [+ "\" contains underscore '_' "]
11     construct Message [id]
12         _   [message Err]
13     construct LenS [number]
14         _ [# ID]
15     construct sub1 [number]
16         US[- 1]
17     construct sub2 [number]
18         US[+ 1]
19     construct ID2 [id]
20         ID[:sub2 LenS]
21     construct NEWID [id]
22         ID[:0 sub1] [+ "$"] [+ ID2]
23     by
24         NEWID
25 end rule
```

Figure 4.6: Replacing Existing Underscores in TXL.

```
1  rule renameTypeRule ModName [id]
2      skipping [type_rule_definition]
3      replace $ [type_rule_definition]
4          ID [id] '::= TYPE [type] ADD [opt scl_additions]
5      construct UniqueRuleName [id]
6          ID [+ '_ ] [+ ModName]
7      by
8          UniqueRuleName '^ ID '::= TYPE [renameType UniqueRuleName] ADD
9  end rule
```

Figure 4.7: Renaming Sequences in TXL.

inside of the FULL_RTPS sequence, and the references used inside the constraint block need to be renamed. In the case of the references made inside the module, all of the declarations that have already been renamed will be checked for validity.

The reference renaming stage begins by renaming all of the imports and exports. The module name from which they are exported is the name appended to the imported type being used in the current module. At this point it is assumed that all import and export names are correct and are renamed without any validation. The validation of import and export names is done later. The renaming continues by building a table of all of the existing declarations, with their long and short names. The TXL script then goes through all of the references inside of the module. The script searches for the old short names in the declarations table when it comes across a reference in the SCL description, and replaces it with the new long name that is used for the declaration. This can be seen in Figure 4.8 for replacing user-defined types inside of a sequence. The replaceSizeType function will check the name pair table NL for an instance where ID, the old short name exists in the table. The new long name, First is then used in the replacement for reference name.

Figure 4.9 shows the final output of the reference renaming phase. The type references that are replaced using the replaceSizeType function are the HEADER_RTPS, GUIDPREFIX_RTPS, and SUBMESSAGE_RTPS types inside of the FULL_RTPS sequence.

Next we describe the renaming of references inside of the constraints blocks. In this example we are focused on the renaming of the references inside of the FULL_RTPS constraint block. Figure 4.10 shows the rules called on each of the expressions inside of the constraint blocks. The renameTransferStatement rule is responsible for finding a reference to a field inside the current sequence, renaming

```
rule renameUserFieldTypes
    import NameList [repeat pair]
    skipping [named_type]
    replace $ [named_type]
        DECL [decl] T [type]
    by
        DECL T [replaceSizeType NameList]
            [replaceSetOfType NameList]
end rule

function replaceSizeType NL [repeat pair]
    replace [type]
        ID [id] SC [size_constraint] OP [opt endian] OPSL [opt slack]
    deconstruct * [pair] NL
        First [id] ID
    by
        First SC OP OPSL
end function
```

Figure 4.8: Searching the name pair table.

```
RTPS DEFINITIONS ::= BEGIN

    EXPORTS PDU_RTPS ^ PDU;

PDU_RTPS ^ PDU ::= (FULL_RTPS | PING_RTPS)

FULL_RTPS ^ FULL ::= SEQUENCE {
    Header_FULL_RTPS ^ Header          HEADER_RTPS (SIZE DEFINED),
    guidPrefix_FULL_RTPS ^ guidPrefix  GUIDPREFIX_RTPS (SIZE DEFINED),
    subMsg_FULL_RTPS ^ subMsg          SET OF SUBMESSAGE_RTPS (SIZE CONSTRAINED)
} (ENCODED BY CUSTOM)
<transfer>
    Back {Header_FULL_RTPS.protoName_HEADER_RTPS == 'RTPS' ||
        Header_FULL_RTPS.protoName_HEADER_RTPS == 'RTPX'}
    Forward {END (subMsg_FULL_RTPS)}
    Callback
</transfer>
```

Figure 4.9: Unique naming output for RTPS sample.

it, and ensuring that any sub-field references are also found and renamed. In the example of the `Header.protoName` reference in Figure 4.9, the dot notation in the constraint is referring to the `protoName` subfield from the `HEADER_RTPS` sequence type, another sequence in the module. The `recurseRename` function is responsible for taking the type `Header_FULL_RTPS` and finding that sequence to look for the declaration of `protoName`. In this case we look at the `HEADER_RTPS` sequence, and look for the short name protoName. If it exists, then we take the long name and rename the sub-field reference. The `renameTransferStatement` will recursively rename references for all nested sub-fields inside the constraints block. The `renameTransferFromRuleDef` rule will rename references to direct user-defined types. The difference in this case is that we are not renaming an instance of a type, but rather the type itself. This can be seen in the following terminating constraint:

```
Forward { TERMINATE(qos) == PIDSENTINAL}
```

There is a reference to the `PIDSENTINAL` type directly, and so all of the sequences in the current module are checked for the existing type before the renaming occurs. By checking that all of the references to declarations exist in the module, we ensure that any references in the constraint blocks actually exist in the module and that the user has not provided a faulty constraints specification. Once all references inside constraint blocks and sequences have been renamed, the type decisions in SCL are processed.

The next part of the reference renaming process renames all of the references inside of type decisions. In the case of the example in Figure 4.4 after the declarations phase, the `PING` and `FULL` references inside of the `PDU_RTPS` need to be renamed. These are renamed in place as they refer to sequences or other type decisions, and

```
rule renameTransferStatement Elist [list element_type]
        Rules [repeat rule_definition]
    skipping [referenced_element]
    replace $ [referenced_element]
        SHORT [id] REST [repeat dot_rp]
    deconstruct * [named_type]  Elist
        FULL [id] '^ SHORT TYPE [type]
    by
        FULL REST [recurseRename TYPE Rules]
end rule

function recurseRename Type [type] Rules [repeat rule_definition]
    replace [repeat dot_rp]
        '. ID [id] REST [repeat dot_rp]
    deconstruct Type
        RTYPE [id] _ [size_constraint]
    deconstruct * [rule_definition] Rules
        RTYPE '^ Short [id] '::= 'SEQUENCE _ [opt size_constraint] '{
            EL [list element_type] _ [opt ',]
        '} _ [opt scl_additions]
    deconstruct * [named_type] EL
        FULL [id] '^ ID TYPE2 [type]
    by
        '. FULL REST [recurseRename TYPE2 Rules]
end function

rule renameTransferFromRuleDef Elist [list element_type]
        Rules [repeat rule_definition]
    skipping [referenced_element]
    replace $ [referenced_element]
        SHORT [id] REST [repeat dot_rp]
    deconstruct * [rule_definition] Rules
        FULL [id] '^ SHORT '::= 'SEQUENCE _ [opt size_constraint] '{
            EL [list element_type] _ [opt ',]
        '} _ [opt scl_additions]
    by
        FULL REST
end rule
```

Figure 4.10: Renaming references from element lists and sequence names.

```
1  function checkExports ModuleName [id]
2      replace [export_block]
3          'EXPORTS DECL [list decl] ';
4      by
5          'EXPORTS DECL [checkExport ModuleName] ';
6  end function
7
8  function checkExport ModuleName [id]
9      replace [list decl]
10         LIST [list decl]
11     construct ExportTable [import_list]
12         LIST 'FROM ModuleName
13     construct outputFile [stringlit]
14         _[+ ModuleName] [+ ".exports"]
15     construct output [import_list]
16         ExportTable [write outputFile]
17     by
18         LIST
19 end function
```

Figure 4.11: Writing exports to disk.

not to specific instances of types. References to imported types from other modules are also renamed in place.

At this point, the unique naming process is finished, and all declarations and references in SCL modules are unique. It is possible that types in different modules end up with identical names. This is only the case specifically when module names are identical in different SCL module definitions. Given that the current prototype is incomplete with regards to future work and input validation, it is assumed that users uniquely name their modules, and for the case of layering network protocols, protocols are uniquely named. Ensuring all module names have a unique name is part of future work and producing a robust semantic phase for analysis of SCL modules.

4.1.3 Export Tables

Immediately following the reference renaming, the module that is being processed will output all of its exports to a separate exports file. This is used later in the SCL analysis to ensure that all imports in modules are legitimate and that an existing definition of that imported module is present and has already been analysed. Figure 4.11 shows the export list being written to a file with the current module name.

4.2 Data Structure Generation

All data types in the forms of sequences and type decisions in SCL need to be represented in the generated parser. C structs are generated and used to hold the parsed data. Figure 4.12 shows the translation from the definition of a DATARSUB sequence in Figure 4.13. Integer types are declared as unsigned integers with the same number of bytes as specified. Integers with odd byte boundaries are rounded up. Integer fields larger than 8 bytes are placed in unsigned character strings. User-defined types are translated to their respective C structs that are generated as seen in line 7 of Figure 4.12. Optional and variable size data like SET OF types are translated into pointers and allocated according to constraints specified at run time. Examples of these translations are shown in section 4.4 pertaining to the source file generation.

The header generation process begins by translating all sequences in place to their respective C structs. All types are determined by checking the number of bytes needed, and assigning a final type accordingly. The generation for type decisions then proceeds. The parser has a choice between multiple user defined types for type decision structs. The data structure generated in this case is a struct

```
typedef struct {
    uint8_t kind;
    uint8_t flags;
    uint16_t nextheader;
    uint16_t extraflags;
    uint16_t qosoffset;
    ENTITYID_RTPS readerent;
    ENTITYID_RTPS writerent;
    uint64_t writerseq;
    QOSPARM_RTPS *inlineqos;
    TOPICS_RTPS *serializeddata;
} DATARSUB_RTPS;
```

Figure 4.12: C struct translation of example in Figure 4.13.

```
DATARSUB ::= SEQUENCE {
    kind       INTEGER (SIZE 1 BYTES),
    flags      INTEGER (SIZE 1 BYTES),
    nextHeader INTEGER (SIZE 2 BYTES),
    extraFlags INTEGER (SIZE 2 BYTES),
    qosOffset  INTEGER (SIZE 2 BYTES),
    readerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
    writerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
    writerSEQ  INTEGER (SIZE 8 BYTES),
    inlineQos  QOSPARM (SIZE DEFINED) OPTIONAL,
    serializedData TOPICS (SIZE DEFINED) OPTIONAL
} (ENCODED BY CUSTOM)
<transfer>
Back {kind == 0x15}
Back {writerEnt.key == 0x4 }
Back {writerEnt.kind == 0xC2 }
Forward { ENDIANNESS == flags & 0x1 }
Forward { EXISTS(inlineQos) == flags & 0x2 }
Forward { EXISTS(serializedData) == flags & 0xC }
</transfer>
```

Figure 4.13: RTPS definition of DATARSUB sequence.

of a union type as seen in Figure 4.14. This structure allows for the static allocation of type decisions and supports one of any of the possible parsing choices. Only one of the types inside of the union will be used, and C will allocate memory for the struct that will match its worst possible case, which is the largest type in one of the four `IGMP` types. While this approach may end up using more memory to hold the data, this structure does not require dynamic memory allocation when compared to a struct of pointers. Figure 4.15 provides the example of statically allocating the struct union type declared in Figure 4.14 and using it. An older version of the parser generator was compiled in C++ to allow structs of pointers to be declared and initialized to `NULL` values. This was required to ensure that there is no misuse of uninitialized pointers. Figure 4.16 shows the example of the older version of the parser generator's structs for type decisions. Figure 4.17 is an example of allocating and using the struct of pointers for one of the types. By using structs of unions, the parser generator avoids having to dynamically allocate memory and minimizes the performance hit associated with constantly allocating memory on the heap. The `type` field in line 7 of Figure 4.14 records which choice a current parse uses.

4.2.1 Supported Types

Other than types that are defined by the user in SCL, we support and generate multiple primitive types depending on what the user specifies in the protocol grammar. All integer types in SCL ranging from one to eight bytes are generated as unsigned integers depending on their size. `uint8_t`, `uint16_t`, `uint32_t`, and `uint64_t` types are used. Octet strings larger than 8 bytes are treated as constant size unsigned chars. Smaller octet strings are translated to their equivalent integer

```
PDU ::= ( V3Report | Query | V2Report | V2Leave)
<transfer>
   Callback
</transfer>

typedef struct {
    uint32_t type;
    union {
        V3Report_IGMP v3report_igmp;
        Query_IGMP query_igmp;
        V2Report_IGMP v2report_igmp;
        V2Leave_IGMP v2leave_igmp;
    } ptr;
} PDU_IGMP;
```

Figure 4.14: C struct union translation for type decision in IGMP.

```
PDU_IGMP pdu_igmp;
if (parseV3Report_0 (&pdu_igmp->ptr.v3report_igmp, thePDU, progname, &type, endianness)) {
    pdu_igmp->type = V3Report_IGMP_VAL;
} else {
    return false;
}
```

Figure 4.15: Struct union static allocation and usage `v3report_igmp`.

```
struct SUBMESSAGE {
    uint16_t    type;
    ACKNACK     *acknack = NULL;
    HEARTBEAT   *heartbeat = NULL;
    GAP         *gap = NULL;
    INFO_TS     *infoTS = NULL;
    INFO_DST    *infoDST = NULL;
    DATASUB     *datasub = NULL;
    DATAWSUB    *datawsub = NULL;
    DATARSUB    *datarsub = NULL;
    DATAPSUB    *datapsub = NULL;
};
```

Figure 4.16: Old C struct for `SUBMESSAGE`.

```
1 //Add link to stack/list
2 SUBMESSAGE *subMsg = (SUBMESSAGE *)malloc(sizeof(SUBMESSAGE));
3 if(subMsg == NULL) {
4     fprintf(stderr,"%s: internal malloc error file: %s line: %d\n",name, __FILE__ , __LINE__);
5     exit(1);
6 }
```

Figure 4.17: SUBMESSAGE allocation for usage.

size. Octet strings that are size constrained are dynamically allocated unsigned character strings, and depend on a length constraint that references a previously parsed field for the size. Real types are also supported. Four-byte real numbers are generated as floats, and eight-byte real numbers are generated as doubles. This covers all of the primitive types that the translator will generate for the custom data structures.

The translations are mostly a one-to-one correspondence between the SCL and the C structures. Figure 4.18 shows the translation to a struct from a sequence type in SCL. The body of the struct is generated for each of the fields inside of the sequence depending on their SCL type as seen in Figure 4.19. Figure 4.20 shows the translation for a field that is a user-defined type, a reference to another sequence in the module.

4.3 LL(1) Optimization

The binary network protocols that are being explored, RTPS, IGMP, and UDP, are context-sensitive and non-ambiguous. This makes it possible to generate recursive-descent parsers with look-ahead capabilities. When the parser reaches part of a PDU with multiple production choices during the translation phase, either an LL(1) look-ahead parser is generated, or a backtracking parser is generated. The parser

```
rule createSequenceStructs
    replace $ [type_rule_definition]
        LONG [id] '^ SHORT [id] '::= 'SEQUENCE OS [opt size_constraint] '{
            LE [list element_type] OC [opt ',]
        '} OP [opt scl_additions]
    construct body [repeat member_declaration]
        _ [createStructBody each LE]
    construct checkCallback [opt scl_additions]
        OP [checkForSequenceCallback LONG]
    by
        'typedef 'struct {
            body
        } LONG ;
end rule
```

Figure 4.18: Translating a sequence in TXL.

```
function createStructBody LE [element_type]
    replace  [repeat member_declaration]
        MD [repeat member_declaration]
    by
        MD [addSizeBasedType LE]
           [addSizeBasedOptionalType LE]
           [addSetOfType LE]
           [addInteger LE]
           [addReal4 LE]
           [addReal8 LE]
           [addDynamicOctetString LE]
           [addStaticOctetString LE]
           [addStaticOctetStringLarge LE]
end function
```

Figure 4.19: Translating single fields inside of a sequence.

that is generated depends on the SCL definition provided by the user for a protocol. The performance of the parser is improved when the optimization can be made.

Sequences that have value constraints on their first field and are part of a type decision can be used to generate an LL(1) look-ahead parser. The value constraints on the first field must all be of the same size in bytes, and all of the values of the field must be mutually exclusive. This is determined when the SCL specification

```
1  function addSizeBasedType LE [element_type]
2      deconstruct LE
3          LONG [id] '^ SHORT [id] TYPE [id] '('SIZE 'DEFINED')
4              OP [opt endian] OPSL [opt slack] POS [opt position_value]
5      construct decl [member_declaration]
6          TYPE SHORT[tolower]';
7      replace [repeat member_declaration]
8          MD [repeat member_declaration]
9      by
10         MD [. decl]
11 end function
```

Figure 4.20: translating a user-defined type.

is being translated into parser source code by examining all sequences specified in a type decision. If the translation engine finds that fields are all of the same size and have different values on the constraints, then a context-sensitive parser is generated for the type decision.

Figure 4.21 shows a partial description of the IGMP module. This portrays the information needed at the input of the LL(1) optimization stage. The `PDU_IGMP` type decision has a choice between the four sequences defined in the IGMP module. This stage will look at the size of the first field, or the `type` field here. It will also check for a value constraint on the `type` field.

The process begins by individually annotating each of the sequences with the size and value of the first field and an `optimizable` annotation. All annotations are then be copied to type decisions where the annotated sequences are referenced. Figure 4.22 shows the result of the LL(1) optimization stage. By analysing the IGMP module and concluding that the `PDU_IGMP` type decision is optimizable, we can generate a non-backtracking parser and for the parsed `type` field and can decide what choice to make based on the value. Figure 4.23 shows the output of the generated source code foe the `PDU_IGMP` type decision.

```
1  PDU_IGMP ^ PDU ::= (V3Report_IGMP | Query_IGMP | V2Report_IGMP | V2Leave_IGMP)
2  <transfer>
3      Callback
4  </transfer>
5
6  Query_IGMP ^ Query ::= SEQUENCE {
7      type_Query_IGMP ^ type INTEGER (SIZE 1 BYTES),
8                  . . .
9  }
10 <transfer>
11     Back {type_Query_IGMP == 17}
12                 . . .
13 </transfer>
14
15                 . . .
16
17 V2Report_IGMP ^ V2Report ::= SEQUENCE {
18     type_V2Report_IGMP ^ type INTEGER (SIZE 1 BYTES),
19                 . . .
20 }
21 <transfer>
22     Back {type_V2Report_IGMP == 22}
23 </transfer>
24
25 V2Leave_IGMP ^ V2Leave ::= SEQUENCE {
26     type_V2Leave_IGMP ^ type INTEGER (SIZE 1 BYTES),
27                 . . .
28 }
29 <transfer>
30     Back {type_V2Leave_IGMP == 23}
31 </transfer>
32
33 V3Report_IGMP ^ V3Report ::= SEQUENCE {
34     type_V3Report_IGMP ^ type INTEGER (SIZE 1 BYTES),
35                 . . .
36 }
37 <transfer>
38     Back {type_V3Report_IGMP == 34}
39                 . . .
40 </transfer>
```

Figure 4.21: Partial IGMP definition, input to LL(1) optimization stage.

```
1 PDU_IGMP ^ PDU @ optimizable ::= (V3Report_IGMP @ 1 34 @ optimizable |
2     Query_IGMP @ 1 17 @ optimizable | V2Report_IGMP @ 1 22
3     @ optimizable | V2Leave_IGMP @1 23 @ optimizable)
4 <transfer>
5     Callback
6 </transfer>
```

Figure 4.22: Partial IGMP definition, output of LL(1) optimization stage.

Sequences that are part of a type decision that do not satisfy the requirement of value constraints on the other hand, will generate a backtracking parser for the type decision. It will attempt to parse each of the production choices in a type decision once. If attempts to parse each of the sequences in a type decision all fail, the overall parse fails.

4.3.1 LL(k) Optimization

There exist type decisions where the first field in the choices do not have mutually exclusive values, and a type decision can still be optimized. An LL(k) optimization can be done when some of the choices in a type decision have other fields inside of the sequence with mutually exclusive value constraints on them. In this case the fields need to be of similar types and at known offsets from the current parse. Figure 4.24 shows the `SUBMESSAGE_RTPS` type decision at the stage when the choices are annotated with the first field values. The `DATA` sub-message types in the type decision all have the same value constraint on the first field. Their `writerEnt` fields however, all have mutually exclusive values as can be seen in Figure 4.25. The markup might be changed in the following manner to support LL(k) for an arbitrary k value. The annotation would add a field offset in bytes where the field has mutually exclusive values, and would follow with thats field's size and value.

```
1  bool parseIGMP (PDU_IGMP *pdu_igmp, PDU *thePDU, char *progname, uint8_t endianness) {
2      if (!lengthRemaining (thePDU, 1, progname)) {
3          return false;
4      }
5      uint8_t type = get8_e (thePDU, endianness);
6      switch (type) {
7      case 34 :
8          if (parseV3Report_0 (&pdu_igmp->ptr.v3report_igmp, thePDU, progname, &type,
9              endianness)) {
10             pdu_igmp->type = V3Report_IGMP_VAL;
11             V3Report_IGMP_callback (&pdu_igmp->ptr.v3report_igmp, thePDU);
12         } else {
13             return false;
14         }
15         break;
16     case 17 :
17         if (parseQuery_0 (&pdu_igmp->ptr.query_igmp, thePDU, progname, &type,
18             endianness)) {
19             pdu_igmp->type = Query_IGMP_VAL;
20             Query_IGMP_callback (&pdu_igmp->ptr.query_igmp, thePDU);
21         } else {
22             return false;
23         }
24         break;
25     case 22 :
26         if (parseV2Report_0 (&pdu_igmp->ptr.v2report_igmp, thePDU, progname, &type,
27             endianness)) {
28             pdu_igmp->type = V2Report_IGMP_VAL;
29             V2Report_IGMP_callback (&pdu_igmp->ptr.v2report_igmp, thePDU);
30         } else {
31             return false;
32         }
33         break;
34     case 23 :
35         if (parseV2Leave_0 (&pdu_igmp->ptr.v2leave_igmp, thePDU, progname, &type,
36             endianness)) {
37             pdu_igmp->type = V2Leave_IGMP_VAL;
38             V2Leave_IGMP_callback (&pdu_igmp->ptr.v2leave_igmp, thePDU);
39         } else {
40             return false;
41         }
42         break;
43     default :
44         return false;
45     }
46     return true;
47 }
```

Figure 4.23: Generated parser for PDU_IGMP type decision.

```
1 SUBMESSAGE_RTPS ^ SUBMESSAGE ::= (ACKNACK_RTPS @ 1 6 | HEARTBEAT_RTPS @ 1 7 |
2     INFO$DST_RTPS @ 1 14 | DATAPSUB_RTPS @ 1 21 | DATASUB_RTPS @ 1 21 |
3     INFO$TS_RTPS @ 1 9 | DATAWSUB_RTPS @ 1 21 | DATARSUB_RTPS @ 1 21 | GAP_RTPS @ 1 8)
```

Figure 4.24: SUBMESSAGE_RTPS type decision that cannot produce an LL(1) parser.

This can be done by checking the output of the LL(1) optimization and analyzing the choices that have the same values for the first field. Once a field that is identical by size has mutually exclusive values in subsequent fields, the optimization phase would add the LL(k) annotation to the choice. There would be a more extensive analysis of the fields to support user-defined types in the example in Figure 4.25, as the TXL program would have to check the `writerEnt`'s sequence for the sub fields `key` and `kind`. The generated parser could look ahead with the offset specified in the annotation to determine which of the choices in the type decision the data is. The LL(k) optimization is a part of the future work for this parser generator framework, and aims to completely remove recursive backtracking from the generated parser for complicated network protocols.

4.4 Source Generation

Constraints are translated directly into the source files that carry out the parsing of the data. Here we provide concrete samples of how the constraints are generated in source code. The examples provided use the same SCL protocol samples used in chapter 3 to describe SCL constraints.

Figure 4.26 shows the parser that is generated as a result of the SCL specification for the sequence in Figure 4.27. All attributes of the C structs generated in the header file will be parsed based on the size of the fields specified in the SCL sequence.

```
1  DATASUB_RTPS ^ DATASUB @ 1 21 ::= SEQUENCE {
2      kind_DATASUB_RTPS ^ kind INTEGER (SIZE 1 BYTES),
3                  . . .
4      writerEnt_DATASUB_RTPS ^ writerEnt ENTITYID_RTPS (SIZE DEFINED) BIGENDIAN,
5                  . . .
6  }
7  <transfer>
8      Back {kind_DATASUB_RTPS == 21}
9      Back {writerEnt_DATASUB_RTPS.kind == 2 || writerEnt_DATASUB_RTPS.kind == 3}
10                 . . .
11 </transfer>
12
13 DATAWSUB_RTPS ^ DATAWSUB @ 1 21 ::= SEQUENCE {
14     kind_DATAWSUB_RTPS ^ kind INTEGER (SIZE 1 BYTES),
15                 . . .
16     writerEnt_DATAWSUB_RTPS ^ writerEnt ENTITYID_RTPS (SIZE DEFINED) BIGENDIAN,
17                 . . .
18 }
19 <transfer>
20     Back {kind_DATAWSUB_RTPS == 21}
21     Back {writerEnt_DATAWSUB_RTPS.key == 3}
22     Back {writerEnt_DATAWSUB_RTPS.kind == 194}
23                 . . .
24 </transfer>
25
26 DATAPSUB_RTPS ^ DATAPSUB @ 1 21 ::= SEQUENCE {
27     kind_DATAPSUB_RTPS ^ kind INTEGER (SIZE 1 BYTES),
28                 . . .
29     writerEnt_DATAPSUB_RTPS ^ writerEnt ENTITYID_RTPS (SIZE DEFINED) BIGENDIAN,
30                 . . .
31 }
32 <transfer>
33     Back {kind_DATAPSUB_RTPS == 21}
34     Back {writerEnt_DATAPSUB_RTPS.key == 256}
35     Back {writerEnt_DATAPSUB_RTPS.kind == 194}
36                 . . .
37 </transfer>
38
39 DATARSUB_RTPS ^ DATARSUB @ 1 21 ::= SEQUENCE {
40     kind_DATARSUB_RTPS ^ kind INTEGER (SIZE 1 BYTES),
41                 . . .
42     writerEnt_DATARSUB_RTPS ^ writerEnt ENTITYID_RTPS (SIZE DEFINED) BIGENDIAN,
43                 . . .
44 }
45 <transfer>
46     Back {kind_DATARSUB_RTPS == 21}
47     Back {writerEnt_DATARSUB_RTPS.key == 4}
48     Back {writerEnt_DATARSUB_RTPS.kind == 194}
49                 . . .
50 </transfer>
```

Figure 4.25: Partial DATA submessage sequences.

```
1  bool parseDATARSUB (DATARSUB_RTPS *datarsub_rtps,
2      PDU *thePDU, char *progname, uint8_t endianness) {
3      if (!lengthRemaining (thePDU, 16, progname)) {
4          return false;
5      }
6      datarsub_rtps->inlineqos = NULL;
7      datarsub_rtps->serializeddata = NULL;
8      datarsub_rtps->kind = get8_e (thePDU, endianness);
9      if (!(datarsub_rtps->kind == 21)) {
10         return false;
11     }
12     datarsub_rtps->flags = get8_e (thePDU, endianness);
13     endianness = datarsub_rtps->flags & 1;
14     datarsub_rtps->nextheader = get16_e (thePDU, endianness);
15     datarsub_rtps->extraflags = get16_e (thePDU, endianness);
16     datarsub_rtps->qosoffset = get16_e (thePDU, endianness);
17     ENTITYID_RTPS readerent;
18     if (!parseENTITYID (&readerent, thePDU, progname, BIGENDIAN)) {
19         return false;
20     }
21     datarsub_rtps->readerent = readerent;
22     ENTITYID_RTPS writerent;
23     if (!parseENTITYID (&writerent, thePDU, progname, BIGENDIAN)) {
24         return false;
25     }
26     datarsub_rtps->writerent = writerent;
27     if (!(datarsub_rtps->writerent.key == 4)) {
28         return false;
29     }
30     if (!(datarsub_rtps->writerent.kind == 194)) {
31         return false;
32     }
33     datarsub_rtps->writerseq = get64_e (thePDU, endianness);
34     if (datarsub_rtps->flags & 2) {
35         datarsub_rtps->inlineqos = (QOSPARM_RTPS *) malloc (sizeof (QOSPARM_RTPS));
36         if (datarsub_rtps->inlineqos == NULL) {
37             return false;
38         }
39         if (!parseQOSPARM (datarsub_rtps->inlineqos, thePDU, progname, endianness)) {
40             free (datarsub_rtps->inlineqos);
41             datarsub_rtps->inlineqos = NULL;
42             return false;
43         }
44     } else {
45         datarsub_rtps->inlineqos = NULL;
46     }
47                 . . .
```

Figure 4.26: Partially generated parser for DATARSUB sequence.

```
1  DATARSUB ::= SEQUENCE {
2      kind       INTEGER (SIZE 1 BYTES),
3      flags      INTEGER (SIZE 1 BYTES),
4      nextHeader INTEGER (SIZE 2 BYTES),
5      extraFlags INTEGER (SIZE 2 BYTES),
6      qosOffset  INTEGER (SIZE 2 BYTES),
7      readerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
8      writerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
9      writerSEQ  INTEGER (SIZE 8 BYTES),
10     inlineQos  QOSPARM (SIZE DEFINED) OPTIONAL,
11     serializedData TOPICS (SIZE DEFINED) OPTIONAL
12 } (ENCODED BY CUSTOM)
13 <transfer>
14 Back {kind == 0x15}
15 Back {writerEnt.key == 0x4 }
16 Back {writerEnt.kind == 0xC2 }
17 Forward { ENDIANNESS == flags & 0x1 }
18 Forward { EXISTS(inlineQos) == flags & 0x2 }
19 Forward { EXISTS(serializedData) == flags & 0xC }
20 </transfer>
```

Figure 4.27: RTPS definition of DATARSUB sequence.

The constraints specified in the SCL definition determine what kinds of checks the parser has to make before completing a parse attempt on the sequence.

4.4.1 Value Constraints

In Figure 4.27 the `kind` field was given a value constraint of 0x15 in hexadecimal, or the decimal value 21. A value constraint in the generated parser will check for the value of the field directly after it is parsed as seen in Figure 4.26 lines 7 and 8. In the case that the `kind` field has a different value, the parse will fail and backtrack. If the caller of the `parseDATARSUB` function is a translation of a type decision from SCL and has been optimized, the parse knows that this sequence type could have been the only possible parse production and will fail parsing the current packet. An unoptimized caller will backtrack and attempt to parse the next sequence type

specified in the type decision.

4.4.2 Nested Constraints

The `writerEnt` field in Figure 4.27 shows a nested value constraint for the DATARSUB sequence. As seen in the generated parser in figure 4.26, a user-defined type requires the parser to call the generated parser for the `ENTITYID` type first. Once the user-defined type has completed the parse successfully, the nested value is checked inside of `writerEnt`. It is possible that in the middle of the `ENTITYID` parsing function that a parse fails. This would be the result of malformed `ENTITYID` data or overall PDU data.

Figure 4.28 shows the translation process for value and nested constraints. Constraint references are translated directly to their C structure name, and the constraint itself is translated as is. The current version of the parser generator will take the conditional expression inside value constraints and keep them in the same format, only renaming the references to fields in the C structures. The value constraint on the `kind` field of the `DATARSUB_RTPS` example would translate to `datarsub_rtps->kind`. This can be seen in line 9 of Figure 4.26, where the the equality condition is the same as the one specified in the SCL description, only that the reference names are substituted with their data structure name.

4.4.3 Endianness

The example used in Figure 4.27 shows that the DATARSUB sequence byte-order depends on the least significant bit of the `flags` field. This implementation interprets a non-zero value as little-endian, and a zero value as big-endian. The

```
1  function checkBackConstraints mallocName [id] RuleName [id] ET [element_type]
2          TR [transfer_statement]
3      deconstruct TR
4          'Back '{ OR [or_expression] '}
5      construct IDS [repeat id]
6          _ [^ OR]
7      deconstruct IDS
8          ID [id] REST [repeat id]
9      deconstruct ET
10         ID '^ SHORT [id] TYPE [type]
11     construct ShortEXP [or_expression]
12         OR [convertStringlitToHex] [createShortExpression]
13     construct ShortIDS [repeat id]
14         _ [^ ShortEXP]
15     deconstruct ShortIDS
16         ShortID [id] ShortREST [repeat id]
17     construct finalEXP [or_expression]
18         ShortEXP [modifyShortExpression ShortID RuleName]
19     construct Stmt [repeat declaration_or_statement]
20         if(!(finalEXP)) {
21             return false;
22         }
23     replace [repeat declaration_or_statement]
24         Stmts [repeat declaration_or_statement]
25     by
26         Stmts [. Stmt]
27 end function
```

Figure 4.28: translating a user-defined type.

constraint in figure 3.4 line 17 forces the the current endianness to change inside of DATARSUB parsing function and carry through to subsequent parsing function calls. This can be seen in line 12 of Figure 4.26. Figure 4.29 shows the change in endianness immediately following the parse of the field that is referenced in the constraints, in this case, the `flags` field. As byte-order is forced to big-endian for the parsing of the `ENTITYID` types as one of the SCL options inside of the DATARSUB sequence, the currently set byte-order will be ignored as seen in line 17 of Figure 4.26. All other sub fields and types being parsed will use the new endianness value. The endianness starts at the top level of the parse, and is passed down as

```
1  function changeEndianness RuleName [id] ET [element_type] OP [opt scl_additions]
2      deconstruct * [transfer_statement] OP
3          'Forward '{ 'ENDIANNESS '== ID [id] '& NUM [number] '}
4      deconstruct ET
5          ID '^ SHORT [id] TYPE [type]
6      construct Stmt [declaration_or_statement]
7          'endianness = RuleName[tolower] '->SHORT[tolower] '& NUM ;
8      replace [repeat declaration_or_statement]
9          Stmts [repeat declaration_or_statement]
10     by
11         Stmts [. Stmt]
12 end function
```

Figure 4.29: Endianness is modified when the ID field is found.

an argument to all of the parsing functions in the source code, effectively creating a stack of endianness values. This allows for the previous endianness value to be properly set if the parser needs to backtrack and attempt a different parse.

4.4.4 Nested Length Constraints

Figure 4.30 provides the example of nested length constraints in SCL. In order to generate the parser code for nested length constraints, it is essential that we avoid the possibility of length overflows in the case of malformed packets. In this example, the length specified in `nameLength` for the `name` string in the `NESTEDSTRING` sequence can be malformed and larger than the total length of the parent field `topicName`. This security vulnerability is avoided by providing the parser of the `NESTEDSTRING` type a size constrained PDU. In Figure 4.31 we take `parameterlength` as the new total possible length of the PDU that is given to the `NESTEDSTRING` parser. In the case that the `nameLength` field of the object is larger than the length of the constrained PDU, the parse will fail. Once the parsing of the nested length is done, we exit the parse and copy new cursor position of the buffer

```
1  PIDTOPICNAME ::= SEQUENCE {
2      parameterKind    INTEGER (SIZE 2 BYTES),
3      parameterLength  INTEGER (SIZE 2 BYTES),
4      topicName        NESTEDSTRING (SIZE DEFINED)
5  }
6  <transfer>
7  Back {parameterKind == 5 }
8  Forward { LENGTH(topicName) == parameterLength }
9  </transfer>
10
11 NESTEDSTRING ::= SEQUENCE {
12     nameLength      INTEGER (SIZE 4 BYTES),
13     name            OCTET STRING (SIZE CONSTRAINED),
14 }
15 <transfer>
16     Forward { LENGTH(name) == nameLength }
17 </transfer>
```

Figure 4.30: Example of nested Length Constraints.

to the PDU that is not constrained. This accounts for the length of the sub length, and the rest of the parent length constraint can continue parsing. This is seen in line 19 of Figure 4.31

Figure 4.32 shows the translation of the `constrainedPDU`. It is generated when the current field being analysed is `SIZE DEFINED` and has a length constraint based on a previously parsed field. Lines 4 and 13 show the TXL script looking for the current field name, and that same reference in the constraints block on a `LENGTH` constraint.

4.4.5 SET OF Fields

`SET OF` fields are essentially lists of single or multiple user-defined types inside of a module. In order to properly generate parsers for `SET OF` fields, the framework requires that one of the three terminating constraints exist for the `SET OF` field. We

```
1           unsigned long pos = thePDU->curPos;
2           if (!lengthRemaining (thePDU, pidtopicname_rtps->parameterlength, progname)) {
3               return false;
4           }
5
6           PDU constrainedPDU;
7           constrainedPDU.data = thePDU->data;
8           constrainedPDU.len = pidtopicname_rtps->parameterlength;
9           constrainedPDU.curPos = pos;
10          constrainedPDU.remaining = pidtopicname_rtps->parameterlength;
11
12          NESTEDSTRING_RTPS topicname;
13          if (!parseNESTEDSTRING (&topicname, &constrainedPDU, progname, endianness)) {
14              return false;
15          }
16          pidtopicname_rtps->topicname = topicname;
17          thePDU->curPos = constrainedPDU.curPos;
```

Figure 4.31: Example of a nested length constraint.

describe them in detail below. The parser generator outputs specific set up code for `SET OF` fields, and also outputs a corresponding `parseSetOf` function that depends on the constraint provided.

Cardinality Terminators

Cardinality terminators specify the number of elements that exist in the `SET OF` list. Figure 4.33 shows the example of the last field being a `SET OF` field in line 5, and a cardinality constraint based on the previously parsed field on line 8. The size of the list in this case is known at runtime, and can directly be referenced in a loop that will call the `SOURCEADDRESS` parse function until the number of items matches the constraint. Figure 4.34 shows the generated setup code to call the `parseSetOfSOURCEADDRESS` function. The count of of the list is determined right after the `numSources` is parsed, and so can be filled in as seen in line 1. The length

```
function checkForwardLengthUserDefined mallocName [id] RuleName [id] OP [opt scl_additions]
        LET [list element_type] ET [element_type]
    deconstruct ET
        LONG [id] '^ SHORT [id] TYPE [id] '(SIZE 'DEFINED)  OPEN [opt endian]
        OPSL [opt slack] POS [opt position_value]
    deconstruct OP
        ENC [opt encoding_grammar_indicator] SZ [opt size_markers_block]
        '<transfer>
        RETR [repeat transfer_statement]
        '</transfer>
        CST [opt constraints_block]
    deconstruct * [transfer_statement] RETR
        'Forward '{ 'LENGTH ( LONG ) '== REST [relational_expression] '}
    replace [repeat declaration_or_statement]
        Stmts [repeat declaration_or_statement]
    construct RESTConverted [relational_expression]
        REST [convertIDToShort] [checkOptionalLengths LET RuleName] [convertSizeOf]
        [convertPOS] [convertPDUREMAINING]
    construct expression [constant]
        RESTConverted [convertToFullID RuleName]
    construct savePos [repeat declaration_or_statement]
        unsigned 'long pos = thePDU ->'curPos;
    construct lengthCheck [repeat declaration_or_statement]
        if(!lengthRemaining(thePDU, expression , progname)) {
            return false;
        }
    construct convertExpression [repeat declaration_or_statement]
        'PDU 'constrainedPDU;
        constrainedPDU.'data = 'thePDU ->'data;
        constrainedPDU.len = expression;
        constrainedPDU.watermark = 'thePDU ->watermark;
        constrainedPDU.curPos = 'pos;
        constrainedPDU.curBitPos = '0;
        constrainedPDU.remaining = expression;
    by
        Stmts [. savePos] [. lengthCheck] [. convertExpression]
end function
```

Figure 4.32: Generating a constrained PDU for nested lengths.

```
1  V3Addition ::= SEQUENCE {
2     resvSQRV     INTEGER (SIZE 1 BYTES),
3     QQIC         INTEGER (SIZE 1 BYTES),
4     numSources   INTEGER (SIZE 2 BYTES),
5     srcAddrs     SET OF SOURCEADDRESS (SIZE CONSTRAINED)
6  }
7  <transfer>
8     Forward{CARDINALITY(srcAddrs)  == numSources}
9  </transfer>
```

Figure 4.33: V3Addition_IGMP SCL definition.

```
1  v3addition_igmp->srcaddrscount = v3addition_igmp->numsources;
2  v3addition_igmp->srcaddrslength = thePDU->curPos;
3  v3addition_igmp->srcaddrs = parseSetOfSOURCEADDRESS (thePDU,
4      v3addition_igmp->numsources, progname, endianness);
5
6  if (v3addition_igmp->srcaddrs == NULL) {
7      return false;
8  }
9  v3addition_igmp->srcaddrslength = thePDU->curPos - v3addition_igmp->srcaddrslength;
```

Figure 4.34: parseSetOf setup code inside parseV3Addition.

of the field is determined once the list is successfully parsed. Figure 4.35 shows the resulting function.

Type Terminators

Type terminators specify what type the last element in the list will be. This constraint is specifically used for sets of type decisions where the total number of elements is unknown, and one of the types in the protocol specification is used as a flag that there are no more elements. This is the case in the RTPS protocol with the `TOPICPARMS` user defined type as seen in Figure 4.36 line 7. It uses the `PIDSENTINAL` type as a flag to terminate the parsing of `topicData`. Figure 4.37 shows the setup code. A size field is initialized to zero and passed into the `parseSetOf` function,

```
1  SOURCEADDRESS_IGMP *parseSetOfSOURCEADDRESS (PDU *thePDU, int size,
2          char *progname, uint8_t endianness) {
3      SOURCEADDRESS_IGMP *result = (SOURCEADDRESS_IGMP *) malloc
4          (size *sizeof(SOURCEADDRESS_IGMP));
5
6      if (result == NULL) {
7          return NULL;
8      }
9      for (int i = 0; i < size; ++i) {
10         SOURCEADDRESS_IGMP srcaddrs;
11         if (parseSOURCEADDRESS (&srcaddrs, thePDU, progname, endianness))
12             result[i] = srcaddrs;
13         else
14             return NULL;
15     }
16     return result;
17 }
```

Figure 4.35: parseSetOf function used in parseV3Addition.

```
1  TOPICS ::= SEQUENCE {
2      encapsKind INTEGER (SIZE 2 BYTES) BIGENDIAN,
3      encapsOpts INTEGER (SIZE 2 BYTES) BIGENDIAN,
4      topicData  SET OF TOPICPARMS (SIZE CONSTRAINED)
5  }
6  <transfer>
7      Forward { TERMINATE(topicData) == PIDSENTINAL}
8  </transfer>
```

Figure 4.36: SET OF TOPICPARMS list has a type terminating constraint.

and is assigned once the size of the list is known following the parse. Figure 4.38 Shows the resulting function that is created for type terminating constraints. A recursive function is created where the next element is attempted to be parsed. Once the last type that has been parsed is of the terminating type, the size is of the list is known, and the list array is allocated and filled in.

```
int size = 0;
topics_rtps->topicdatalength = thePDU->curPos;
topics_rtps->topicdata = parseSetOfTOPICPARMS_O (thePDU, 0, &size, progname, endianness);
if (topics_rtps->topicdata == NULL) {
    return false;
}
topics_rtps->topicdatacount = size;
topics_rtps->topicdatalength = thePDU->curPos - topics_rtps->topicdatalength;
```

Figure 4.37: parseSetOfTOPICPARMS_O setup code.

```
TOPICPARMS_RTPS *parseSetOfTOPICPARMS_O (PDU *thePDU, int n, int *size,
    char *progname, uint8_t endianness) {
    TOPICPARMS_RTPS topicdata;
    if (parseTOPICPARMS (&topicdata, thePDU, progname, endianness)) {
        if (topicdata.type == PIDSENTINAL_RTPS_VAL) {
            TOPICPARMS_RTPS *result = (TOPICPARMS_RTPS *) malloc
                ((n + 1) * sizeof (TOPICPARMS_RTPS));
            if (result == NULL) {
                return NULL;
            }
            *size = n + 1;
            result[n] = topicdata;
            return result;
        } else {
            TOPICPARMS_RTPS *result = parseSetOfTOPICPARMS_O (thePDU,
                n +1, size, progname, endianness);
            if (result != NULL) {
                result[n] = topicdata;
            }
            return result;
        }
    } else {
        if (n == 0) {
            *size = 0;
            return NULL;
        } else {
            return NULL;
        }
    }
}
```

Figure 4.38: parseSetOf function used in parsing TOPICPARMS_RTPS list.

End Terminators

End terminators are constraints on `SET OF` fields that are at the end of a PDU, and the size of the list is unknown. This is usually present in fields that are a list of user defined types that have variable sizes. Similar to terminating constraints, the size at the beginning of the field parsing is unknown, and so a recursive function is generated that will attempt to parse the next type in the list. The difference here however is that the function will keep trying to parse the `SET OF` type. Once a parse fails, the function will backtrack to the previously parsed data and allocate the list and return. After the list has been created, the PDU will be checked for whether there exists any remaining data. If there is no data present following an end terminating constraint, then the parse succeeds. This type of constraint is needed in complex protocols, where the number of elements in a list field is not given inside the PDU. This is present in the RTPS protocol, specifically for the `SUBMESSAGE_RTPS` type, where the field `subMsg SET OF SUBMESSAGE (SIZE CONSTRAINED)` has an ending terminator `Forward {END (subMsg}`

4.4.6 Slack

There exists a concept of slack bytes in fields with length constraints. Due to byte alignment issues and protocol field sizes, certain fields may introduce bytes that follow a sequence parse in order to ensure bytes are aligned inside of a PDU. This issue arises in more complex network protocols like RTPS where data types have nested length constraints and bytes remain, or bytes need to be aligned. There are two distinct slack options in SCL that the user can specify in order to skip slack bytes. The first is specifying that slack bytes exist after a field has been parsed as

```
1  PIDMULTICASTLOCATOR ::= SEQUENCE {
2      parameterKind    INTEGER (SIZE 2 BYTES),
3      parameterLength  INTEGER (SIZE 2 BYTES),
4      locator          LOCATOR (SIZE DEFINED) SLACK
5  }
6  <transfer>
7      Back {parameterKind == 48 }
8      Forward { LENGTH(locator) == parameterLength }
9  </transfer>
10
11 LOCATOR ::= SEQUENCE {
12    kind     INTEGER (SIZE 4 BYTES),
13    port     INTEGER (SIZE 4 BYTES),
14    address  OCTET STRING (SIZE 16 BYTES)
15 }
```

Figure 4.39: SLACK specified for length constrained field.

seen in Figure 4.39 line 4. This slack option can only occur when there exists a length constraint on a user defined field, in this case `locator`, and the size of the type itself is smaller than the given constraint size. Once the `locator` field has been parsed, the parser generator ensures that the remaining bytes for `PIDMULTICASTLOCATOR` are skipped. This will generate code that ignores the remaining bytes after parsing a constrained PDU.

The second slack type is specifying that a field automatically needs to be aligned every four bytes when length constraints are involved. This can be seen in Figure 4.40 lines 2, 3 and 11. In this case the parser does not know the size of the `NESTEDSTRING` types until run-time when the `nameLength` field is parsed. In this case no constrained PDU is involved, and once a string is parsed, the data structure is aligned on a four byte boundary. The parser calculates the closest four byte boundary, and skips the remaining number of bytes before continuing a parse.

```
1  PROPERTY ::= SEQUENCE {
2      name    NESTEDSTRING (SIZE DEFINED) SLACKMOD4,
3      value   NESTEDSTRING (SIZE DEFINED) SLACKMOD4
4  }
5
6  NESTEDSTRING ::= SEQUENCE {
7      nameLength      INTEGER (SIZE 4 BYTES),
8      name            OCTET STRING (SIZE CONSTRAINED),
9  }
10 <transfer>
11     Forward { LENGTH(name) == nameLength }
12 </transfer>
```

Figure 4.40: Four byte-aligned slack specification.

4.5 Import/Export Validation

Once all modules have been fully analysed and generated, there is a validation phase for all modules that have imported other module types. This is usually the case at every new layer in the networking model. In the case of the NIDS project, the current module that imports other protocol modules is the UDP module. The sole purpose of the import validation is to read all of the export files that match an import module name, and check the export list in that module for all of the imports. Any modules that are not found in the export tables are reported to the user and the translation phase will exit with an error listing the undefined imports. This avoids the user from making syntactical errors in the imports and ensures that the generated parser for all modules will run correctly.

Chapter 5

Evaluation

The parser generator that was created achieves the five goals described in section 1.2 due to its design. We have picked SCL as a specification language due to its similarity to ASN.1 and binary protocol specifications in RFCs. We have also introduced the different constraint types and extensions to the SCL language that are needed to create the recursive-descent LL(1) parsers. By automating the parser generation, we are able to ensure that parsers are secure and perform at high speeds in C source code. This section discusses the build system, validity, and performance of the generated parsers.

5.1 Build System

It is important to note that the parser generator framework's build system requires minimal dependencies. The translation part of the framework itself only requires the TXL application to be installed. TXL is multi-platform. The project requires no operating system dependent libraries for the parser generation and use in live networking sessions. The generated code itself is multi-platform, and is currently

compiled using the GNU GCC [13] compiler. There are no compiler dependent extensions used in any of the generated code, and it is entirely self-contained. It is be possible to compile this using any standard C99 compiler including Clang [10] and MinGW [28]. The current build of the system has been extended to support the offline testing of network packets, which requires the libpcap library [33]. This can be compiled on Unix systems supporting libpcap. A part of future goals for the generated parser is to support opening raw sockets directly to use the parser for the NIDS in a live system. All of the current evaluation has been done offline, using the libpcap library. To avoid the bottleneck of disk I/O while reading offline packet captures. All tests have been done on solid state drives, where the throughput of the devices is faster than the current speed of the parser. Ram disks were used to ensure that the solid state drives used in fact do not slow down the performance of the parser prior to continuing tests using the drives.

The translation engine itself including the grammar that defines the syntax of SCL and the TXL transformation scripts is a total of 6120 lines of code. The translation process is light-weight and minimal. The size of the project is small when compared to other parser generator frameworks. This makes the generator framework easy to maintain and debug.

5.2 Packet Validity Testing

The parser generator will attempt to inspect all of the data in a single packet. All attributes defined in the SCL specification must be matched in the real data, otherwise the full packet is flagged as invalid. All malformed data has been created by taking valid packet captures and modifying specific fields in the data using a

hexadecimal editor.

5.2.1 Parser Success

In order to ensure that protocols specified in SCL and translated into generated parsers are complete, we use test data from our experimental network [25]. This network is a set of virtual machines that run a flight simulation where one machine acts as the server, which generates radar data and spawns multiple aircraft with their own data. Another machine runs the air traffic controls software and communicates with each of the flights. Other machines exist on the network to generate network traffic such as NTP. The air traffic control software has been extended to communicate using RTPS and IGMP. The simulations provide real world data resembling air traffic control systems. The protocols on the experimental network are RTPS, NTP, IGMP, ARP, and TCP. The protocols that are currently supported by the generated parser are RTPS, NTP, IGMP, and ARP.

The process to support a network protocol begins by finding the protocol specification as defined, and implementing it in SCL. Constraints are filled in to ensure that protocols are fully defined for the parser. Packet captures are filtered for the one protocol being tested. The filtered captures are then used to test a parser that is generated for this protocol. The packet captures used are engineered to contain all of the different possible data types in one specific protocol and only contain valid data. These captures are then used to test the parser until none of the packets fail to parse. Once an SCL specification is complete and will completely parse a protocol, it is added to the NIDS system used for the project.

As the NIDS system is supposed to generate alerts when certain attacks are

made, the system needs to be able to differentiate between the types of attacks when informing network administrators. the parser is only responsible for the validity of packets and will generate alerts for when a packet fails to parse. Packet validity can be tested once the generated parsers have been checked for completeness.

5.2.2 Overflow Protection

There are two mechanisms in the design of the generated parser that avoid the possibility of buffer overflows in the parser itself. The first mechanism is that the lengths of user-defined types are always determined before parsing the attributes of a sequence. All primitive type sizes are aggregated, and the value is used to check the PDU buffer for enough data. If enough data is not available before a parser, the packet will fail the validity check. Primitive or non primitive types that have lengths determined at run time are aggregated according to the data provided in the PDU. Once these are determined, the PDU is also checked for remaining data before parsing.

The parser generator avoids overflows in the actual generation in two ways. First, when any data structure generated involves a pointer value that must be allocated at run time, all pointers are initialized to `NULL` as soon as they are accessible in the parsing code itself. This avoids the possibility of any dangling pointer vulnerabilities. All pointers are also always checked whether they are `NULL` or not before access, ensuring that pointer values have either been filled in, or not, and avoids the chance of accessing random memory location through unused pointers. All pointers are checked in the garbage collection phase of the packet for example, and without the generator initializing pointers to null, could allow for

random memory access. The second mechanism that is built into the parsers to avoid buffer overflows are the constrained PDU's. Constrained PDU's are always created when length constraints are nested. By providing a constrained PDU for a nested length, sub-length constraints that are larger than parent length constraints will generate an error and fail the parse of the packet. Table 5.1 shows the results between running a clean packet capture of RTPS data using the generated parser, and the same packet capture with nested length overflows in child length fields of five different packets. The Data Distribution Service's network protocol RTPS, is the only protocol in our network test data that hold run time length constraints that are nested.

Table 5.1: Nested Length Overflow of Five Packets

Capture Data	Capture Size (MB)	Total Packets	Failed Parse
Clean RTPS	1,492	10,518,612	0
Malformed RTPS	1,492	10,518,612	5

Due to the parser being automatically generated from the specification, human error in these types of validity checks are not present as opposed to hand written parsers. Protocols that are complicated in nature like RTPS may require upwards of 5,000 lines of code. Manually written parsers are regularly prone to error when establishing packet validity mechanisms.

5.3 Performance

We evaluate the generated parser by comparing to the performance of the SCL implementation described by Marquis et al. [22]. The SCL parser is a universal

interpretive parser that is used for penetration testing. It is too slow to parse network data for a real-time NIDS system that is required to run at speeds supported by modern routers and switches. Here we compare the speeds of both parsers by running packet captures of both isolated and mixed network protocols. All tests were done on a dual-core cpu running at 2.5 Ghz.

Tables 5.2 and 5.3 shows the performance differences between the interpretive parser and the generated parser. These were testing using identical packet captures containing RTPS data.

Table 5.2: Interpretive Parser Performance

Capture Size (MB)	Run Time (s)	Bandwidth (Mbit/s)
77	15.1	40.79
225	33.9	53.10
962	148.8	51.74

Table 5.3: Generated Parser Performance

Capture Size (MB)	Run Time (s)	Bandwidth (Mbit/s)
77	0.2	3,080.00
225	0.7	2,571.43
962	2.7	2,850.37

While the results shown do not model a realistic network, they show a significant increase in performance by taking the parser generation approach. This is due to the generator creating custom dissectors for each type of protocol. The separate parsers are grouped into one parser and can be used on network data containing the protocols defined in SCL by the user.

Table 5.4 shows the result of using the generated parser on more realistic network

data. The packet captures generated have been used to test the constraint-based NIDS system being built by our research group. The captures are generated using the RTI version of the Data Distribution Service (DDS) [17] to generate RTPS data. The protocols that are captured on the network are IGMP, RTPS, ARP, and NTP. All of this data is generated as part of an air traffic control simulation to model real-world data.

Table 5.4: Realistic Parser Performance

Capture Size (MB)	Run Time (s)	Bandwidth (Mbit/s)
1,668	11.5	1,191.43
3,336	22.0	1,213.09
6,672	46.3	1,152.83

The average bandwidth support sustained by the unified generated parser is just above gigabit speeds. This would allow the parser to be used in real-time networks with switches that support gigabit speeds, and can be used as the dissector for an NIDS on a live network. These results include minimal optimizations to the parser generator other than the LL(1) optimizations inferred from the SCL constraints. There has been an extensive list of similar work done in the research community, and we discuss these in the following section.

Chapter 6

Future Work and Conclusions

While significant progress has been made in providing an extended version of SCL to generate fast parsers, there are multiple optimizations and additions that can be made to the parsing framework.

6.1 Future Work

6.1.1 TCP Stream Assembly and Parsing

The framework in its current state is only able to examine binary data and cannot generate parsers for stream-based data. This eliminates the ability to parse any TCP protocols on the network, and limits parser capabilities. The parser generator's main goal in future work is TCP stream assembly and the parsing of connection-based protocols. This would allow users to parse most network protocols that exist today using SCL descriptions.

6.1.2 LL(k) Optimization

There exist type decisions in SCL that cannot be optimized to allow for the generation of an LL(1) parsers. These are due to two possibilities. The first is that no value constraints are specified on the first field of all sequences that exist inside of a type decision. The second is that all the sequences in a type decision have value constraints; some of which the values are not mutually exclusive. The latter instance includes sequences that have identical values for their first field, and different values for subsequent fields that differentiate the type of a sequence. By checking value constraints on multiple fields in a sequence, an LL(k) optimization can be made, and the generated parser will not need to backtrack in these instances.

6.1.3 Debugging Framework

A user is required to use a source level debugger like GDB [14] in order to determine why a parse fails. The process is time consuming and requires that users understand the generated parsers at the source code level. An extension to this project would not only notify users which packets have failed to parse, but where in the SCL specification the packet could not continue to parse. This would allow users to quickly identify whether there is information missing in the SCL description they are using, or whether a packet is actually malformed.

6.1.4 Semantic SCL Validation

In its current form, the framework checks for valid SCL syntax. The translation framework however does not check all of the semantic information written by the user. For example, an `OPTIONAL` field must have a corresponding `EXISTS` constraint

for the generator to properly create the parser. A list field of the form `SET OF` must also be accompanied by a terminator constraint. References to other sequences inside the constraints block of the current sequence need to be valid. These types of checks are not made by the current framework. It is essential that a semantic validation is added to the generator framework to minimize user error and incorrect parser generation.

6.1.5 Fuzz Testing

The parsers generated from the translation framework have been tested for vulnerabilities through the carefully placed buffer overflows and invalid data in know valid packet captures. The system itself however has not been fully tested for any obscure vulnerabilities introduced in the generation of the parsers. Part of the future work involves ensuring that the overall NIDS system cannot be used by an attacker to compromise the network through the analysis system itself. It is therefore imperative that the NIDS itself is not vulnerable. It has been proposed that both the network parser and the constraint engine of the NIDS system be fuzz tested with random data.

6.2 Conclusion

In this thesis, we have suggested a new approach to deep packet inspection by using a domain specific language and parser generation to tackle the problem of low-performance binary parsers. Developers that wish to use this approach will spend significantly less time by writing SCL descriptions rather than custom source

code to create their specialized network parsers. They will also achieve the same performance as hand written dissectors without the need of spending time ensuring that dissectors are secure and reliable. The SCL descriptions can be written to parse certain sections, or all data inside a set of network packets. They can also be used to specify value constraints and allow the generation of LL(1) look-ahead parsers. We have provided a modular approach that allows users to determine what layers in the networking model they wish to dissect, and without extensive optimization, have provided a framework that generates high performance dissectors that are robust.

The parser requirements have all been achieved with the initial prototype. These are summarized below.

6.2.1 High-Performance

Full deep-packet inspection of network packets has been achieved on realistic network data with generated code similar to hand written parsers. Throughputs that are higher than those of gigabit switches have been achieved, allowing the generated parsers to run on high-speed industrial networks that support gigabit networks.

6.2.2 Reliability

The system has been tested for memory leaks, and it has been ensured that the parser generator will not generate code that causes memory leaks in the system. This allows for no down time or crashes due to memory usage. All parses that fail will also be written to trace files and the parse will exit gracefully. The parser itself

continues by parsing the rest of the network data.

6.2.3 Security

The parser generator's code templates were created to check for the possibility of buffer overflows. Parts of packets that depend on lengths are taken into account to ensure that sub-lengths of packet data cannot be greater than parent lengths. All declared and freed pointers are assigned to null, removing the possibility of dangling pointers. Packets with unintended data according to protocol specifications will fail to parse and be flagged. All of these mechanisms are also automatically generated, removing the possibility of human error when writing custom parsers. The error is only possible at the code template level. The code templates have been tested and designed specifically to avoid common security pitfalls that are often seen in hand written code. these mechanisms also aid in ensuring that the NIDS system itself cannot be attacked by providing it malformed data.

6.2.4 High-Level Specification

The extended SCL language is being used specifically to provide both a high-level specification language and input to the function of the NIDS, and to provide a familiar language for network engineers. The specification language will also be used to generate the constraint engine code for the NIDS in the future, and can be used for the whole system. This requirement is one of the main reasons this project does not rely on other related work to achieve the goal of the generated parsers.

6.2.5 Modular

Modularity is built-in to ASN.1 and SCL, allowing modules to be imported, and providing an interface to layer networking protocols. The modularity of SCL has been kept in the translation pipeline, and modules can be imported and used. The generated code layers the modules as specified by the user. Our current prototype includes layered modules in UDP, NTP, and RTPS.

The main contributions achieved have been extending the SCL language to support the creation of context-sensitive parsers, and creating the parser generator framework using a new approach to solve an problem that is consistent across the network traffic analysis community.

Bibliography

[1] Muhammad AboElFotoh, Thomas Dean, and Ryan Mayor. An empirical evaluation of a language-based security testing technique. In *Proceedings of the 2009 conference of the Centre for Advanced Studies on Collaborative Research, November 2-5, 2009, Toronto, Ontario, Canada*, pages 112–121, 2009.

[2] Julian Bangert and Nickolai Zeldovich. Nail: A Practical Interface Generator for Data Formats. In *35. IEEE Security and Privacy Workshops, SPW 2014, San Jose, CA, USA, May 17-18, 2014*, pages 158–166, 2014.

[3] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol. `https://tools.ietf.org/html/rfc1157`. Accessed: 2017-05-29.

[4] James R. Cordy. The TXL Source Transformation Language. *Sci. Comput. Program.*, 61(3):190–210, August 2006.

[5] James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider. Source transformation in software engineering using the TXL transformation system. *Information & Software Technology*, 44(13):827–837, 2002.

[6] Thomas R. Dean, James R. Cordy, Kevin A. Schneider, and Andrew J. Malton. Using Design Recovery Techniques to Transform Legacy Systems. In *2001*

International Conference on Software Maintenance, ICSM 2001, Florence, Italy, November 6-10, 2001, pages 622–631, 2001.

[7] O. Dubuisson. *ASN.1 Communication Between Heterogeneous Systems*. Morgan Kaufmann, 2001.

[8] Ali ElShakankiry. alishak/TAG: TAG - The Traffic Analysis Generator Framework. `https://github.com/alishak/TAG`. Accessed: 2017-08-21.

[9] B. Fenner, H. He, B. Haberman, and H. Sandick. Internet Group Management Protocol. `https://tools.ietf.org/html/rfc4605`. Accessed: 2017-05-29.

[10] The LLVM Foundation. clang: a C language family frontend for LLVM. `http://clang.llvm.org/`. Accessed: 2017-07-3.

[11] The LLVM Foundation. The LLVM Compiler Infrastructure. `http://llvm.org/`. Accessed: 2017-07-2.

[12] The Wireshark Foundation. Wireshark Security Advisories. `https://www.wireshark.org/security/`. Accessed: 2017-05-29.

[13] Inc. Free Software Foundation. GCC, the GNU Compiler Collection. `https://gcc.gnu.org/`. Accessed: 2017-07-3.

[14] Inc. Free Software Foundation. The GNU Project Debugger. `https://www.gnu.org/software/gdb/`. Accessed: 2017-05-29.

[15] The Open Group. Distributed Relational Database Architecture (DRDA) Standard. `https://collaboration.opengroup.org/dbiop/`. Accessed: 2017-05-29.

[16] Md Siam Hasan, Ali ElShakankiry, Thomas Dean, and Mohammad Zulkernine. Intrusion detection in a private network by satisfying constraints. In *14th Annual Conference on Privacy, Security and Trust, PST 2016, Auckland, New Zealand, December 12-14, 2016*, pages 623–628, 2016.

[17] Real-Time Innovations. RTI. `https://www.rti.com/products/dds`. Accessed: 2017-05-29.

[18] ITU. ASN.1. `http://www.itu.int/itu-t/recommendations/rec.aspx?rec=x.680`. Accessed: 2017-05-29.

[19] D. Joyal, P. Galecki, S. Giacalone, R. Coltun, and F. Baker. OSPF Version 2 Management Information Base. `https://tools.ietf.org/html/rfc4750`. Accessed: 2017-05-29.

[20] Nicolas Laurent and Kim Mens. Taming context-sensitive languages with principled stateful parsing. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 15–27, New York, NY, USA, 2016. ACM.

[21] Sylvain Marquis, Thomas R. Dean, and Scott Knight. SCL: a language for security testing of network applications. In *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative Research, October 17-20, 2005, Toronto, Ontario, Canada*, pages 155–164, 2005.

[22] Sylvain Marquis, Thomas R. Dean, and Scott Knight. Packet decoding using context sensitive parsing. In *Proceedings of the 2006 conference of the Centre for*

Advanced Studies on Collaborative Research, October 16-19, 2006, Toronto, Ontario, Canada, pages 263–274, 2006.

[23] Sylvain Emmanuel Marquis. SCL: A generic technique for specifying semantic constraints between fields of a network message for vulnerability testing. Royal Military College of Canada, 2010.

[24] D. Mills, U. Delaware, Ed. J. Martin, J. Burbank, and W. Kasch. Network Time Protocol Version 4: Protocol and Algorithms Specification. `https://tools.ietf.org/html/rfc5905`. Accessed: 2017-05-29.

[25] L.-P. Morel. Using Ontologies To Detect Anomalies In the Sky. Département de génie informatique et génie logiciel, Expected 2017.

[26] Serkan Özkan. Wireshark Vulnerability Trends Over Time. `https://www.cvedetails.com/product/8292/Wireshark-Wireshark.html?vendor_id=4861`. Accessed: 2017-05-29.

[27] Terence Parr and Kathleen Fisher. LL(*): the foundation of the ANTLR parser generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 425–436, 2011.

[28] Colin Peters, Jan-Jaap van der Heijden, Mumit Khan, Anders Norlander, Earnie Boyd, and Dale Handerson. MinGW — Minimalist GNU for Windows. `http://mingw.org/`. Accessed: 2017-07-3.

[29] David C. Plummer. An Ethernet Address Resolution Protocol. `https://tools.ietf.org/html/rfc826`. Accessed: 2017-05-29.

[30] J. Postel. User Datagram Protocol. `https://tools.ietf.org/html/rfc768`. Accessed: 2017-05-29.

[31] Robin Sommer, Johanna Amann, and Seth Hall. Spicy: a unified deep packet inspection framework for safely dissecting all your data. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, pages 558–569, 2016.

[32] Robin Sommer, Matthias Vallentin, Lorenzo De Carli, and Vern Paxson. HILTI: an abstract execution environment for deep, stateful network traffic analysis. In *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*, pages 461–474, 2014.

[33] Tcpdump/Libpcap. TCPDUMP/LIBPCAP public repository. `http://www.tcpdump.org/`. Accessed: 2017-07-3.

[34] Inc. Upstanding Hackers. Hammer. `https://github.com/abiggerhammer/hammer`. Accessed: 2017-05-29.

[35] Songtao Zhang, Thomas R. Dean, and Scott Knight. A lightweight approach to state based security testing. In *Proceedings of the 2006 conference of the Centre for Advanced Studies on Collaborative Research, October 16-19, 2006, Toronto, Ontario, Canada*, pages 341–344, 2006.