2017 15th Annual Conference on Privacy, Security and Trust

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

Ali ElShakankiry
Queen's University School of Computing
Kingston, ON, Canada
0sae@queensu.ca

Thomas Dean
Department of Electrical and Computer Engineering
Queen's University
Kingston, ON, Canada
tom.dean@queensu.ca

Abstract—**Network protocol parsers constantly dissect a large number of packets to place into internal data structures for further processing. We propose an approach that automatically generates custom protocol parsers to process network traffic to be used as part of an Intrusion Detection System. This paper takes a look at the case of 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.**

1. Introduction

Network protocol parsers are extensively used 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 [1], [2]. 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. [3]. 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.

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 [4], IGMP [5], and multiple UDP [6] protocols.

1.1. Parser Requirements

A network-based intrusion detection system needs to ensure that there is no malicious network data. It is essential that its parser operates with the following requirements in mind. The parser needs to support the maximum throughput of the network. It should also 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. This 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 the generated parser achieves these goals.

Performance: The generated custom parsers for the example UDP based protocols is able to parse packets at 1.2 GBits per second in a single thread, the speed of the many limited networks.

Reliability: Since the parser is generated from a high level specification, the parser implementation contains all of the checks, such as the release of unused memory, that might be omitted in a manually written parser.

Security: The parser generator's code templates were created to check for the possibility of buffer overflows. Packets with unintended data according to protocol specifications will fail to parse.The code templates have been tested and designed specifically to avoid common security pitfalls that are often seen in hand written code.

High-Level Specification: ASN.1 notation was introduced by the ITU [7], [8] to specify binary protocols. By extending ASN.1, Syntax Constraint Language [9] allows the user to specify parsing constraints utilizing a notation that is created specifically for internet protocols like OSPF [10] and SNMP [11].

Modular: SCL takes advantage of the ANS.1 module concept. Each protocol is written in a separate SCL module, and combined to form a single parser for the group of protocols for a particular frame type. The grammars may

978-1-5386-2487-6/17/$31.00 ©2017 IEEE
DOI 10.1109/PST.2017.00019

also be layered allowing a single parser to be generated for all of the defined network protocols.

The structure of the following sections in the paper is as follows. Section 2 provides a description of the backbone of our DPI framework. Sections 3 and 4 describe the input and output to the parser generator framework in depth. The evaluation follows in Section 5, and we discuss related works in section 6. We end this paper by laying out plans for future work and concluding in Section 7.

2. Background

2.1. 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 [12], [13]. It has been used in solving large-scale real world problems involving billions of lines of code such as the year 2000 problem [14]. 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.

3. Syntax Constraint Language

SCL provides the network security specialist a means to define a modular syntax specification for the binary protocol they wish to parse. SCL is an extension to ASN.1, and is used due to its ease of defining network protocols. The extensions developed by Marquis et al. 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 and provides the information needed to generate optimized parsers. SCL has been used before to test the security of network protocol data by mutating packets and reassembling them to wire format data [15], [16].

In this section we discuss the importance of adapting ASN.1 to create a parser that is modular in addition to the constraints added by the SCL specification to allow context-sensitive parsing.

```
NTPV4 DEFINITIONS ::= BEGIN

   EXPORTS PDU;

PDU ::= SEQUENCE {
   flags            INTEGER (SIZE 1 BYTES),
   peerStratum      INTEGER (SIZE 1 BYTES),
   peerInterval     INTEGER (SIZE 1 BYTES),
   peerPrecision    INTEGER (SIZE 1 BYTES),
   rootDelay        INTEGER (SIZE 4 BYTES),
   rootDispersion   INTEGER (SIZE 4 BYTES),
   referenceId      INTEGER (SIZE 4 BYTES),
   referenceTS      INTEGER (SIZE 8 BYTES),
   originTS         INTEGER (SIZE 8 BYTES),
   recieveTS        INTEGER (SIZE 8 BYTES),
   transmitTS       INTEGER (SIZE 8 BYTES)
} (ENCODED BY CUSTOM)
<transfer>
   Back{ (flags & 56) == 32 }
</transfer>

END
```

Figure 1. Defining NTP version 4 in SCL.

3.1. ASN.1

In this section we describe how the ASN.1 specification is used in SCL to keep the generated parser modular.

Figure 1 shows an NTPV4 module defined in SCL that exports its top level object. In this case the full protocol data unit or PDU for the Network Time Protocol [17]. Exporting PDU from the NTPV4 module allows the UDP module in Figure 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 will not generate any layer 4 parsing for the UDP protocol. In this case, an end user has to manually skip the UDP header at layer 4 to pass the data properly 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 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 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 the data for which they want to generate a parser.

```
UDP DEFINITIONS ::= BEGIN
    IMPORTS PDU FROM RTPS, PDU FROM NTPV4;
PDU ::= (RTPS.PDU | NTPV4.PDU)
END
```

Figure 2. Unifying modules that run on top of the UDP protocol.

```
UDP DEFINITIONS ::= BEGIN
    IMPORTS PDU FROM RTPS, PDU FROM NTP;
PDU ::= SEQUENCE {
    srcPort     INTEGER (SIZE 2 BYTES),
    dstPort     INTEGER (SIZE 2 BYTES),
    length      INTEGER (SIZE 2 BYTES),
    checksum    INTEGER (SIZE 2 BYTES),
    udpProto    UDP_PROTO (SIZE DEFINED)
} (ENCODED BY CUSTOM)

UDP_PROTO ::= (RTPS.PDU | NTP.PDU)
END
```

Figure 3. Including UDP layer 4 parsing for the generator.

3.2. 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 that can be appended to type decisions and sequences, lines 3 and 5 in Figure 4, respectively.

3.2.1. Custom Extensions.

3.2.1. 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 4 line 6. This avoids having to take the approach introduced by Marquis et al. [9] 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 are described in Section 4.

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 4 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 10 of Figure 4, 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.

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

Figure 4. Partial IGMP definition with SCL constraints.

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

Figure 5. RTPS definition of TOPICS sequence.

3.2.2. 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 23 in Figure 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 7 unless otherwise specified.

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

Figure 6. partial listing of the TOPICPARMS type decision.

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 7 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. The parser can also decide what the next sequence type to be parsed is going to be. This can only be done when there exists value constraints in multiple sequences for the same attribute type. The attribute in this case must be at the beginning of the sequences, like `kind` in this example. This optimization is discussed in section 4.4 and allows the generation of the LL(1) parser.

Nested Constraints. A nested constraint is a value constraint where the value that needs to be checked is inside of a sub-type. In the case of line 15 in Figure 7, the `key` data 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 7 provides the example of nesting the endianness of data in the PDU. Line 17 in Figure 7 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.

Terminators. Figure 5 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 5, 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 5 says that the last user-defined type that should be in the list of topicData is the PIDSENTINAL type, a child of the TOPICPARMS type as seen in Figure 6. 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 24 of Figure 4, 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 ) }
```

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.

```
 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 7. RTPS definition of DATARSUB sequence.

```
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 8. C struct translation of example in Figure 7.

Length Constraints. Length constraints exist when the length of a field depends on a previously parsed value. Figure 10 shows two sequences that both have length constraints. The `NESTEDSTRING` sequence is used as the type for the `topicName` field in the `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 the next section.

4. Parser Generation

This section provides a brief overview of the resulting parser code that is generated. This is done by taking the SCL-defined protocols and translating them into C code. The output for each protocol is two files. The first is a header file that defines all the C structures used to fill in parsed data. The second is a source file that parses data from the PDU given to verify that the data is according to the SCL specification given by the user. This can be seen in Figure

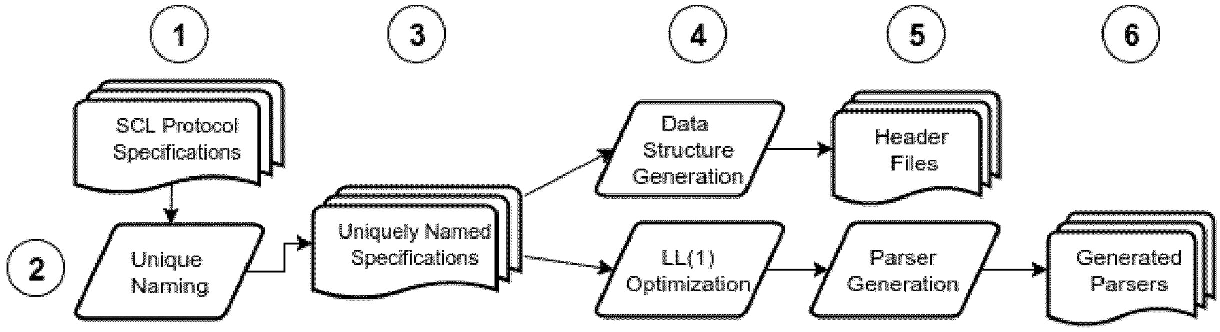

Figure 9. Parser generation flow chart.

```
PIDTOPICNAME ::= SEQUENCE {
    parameterKind    INTEGER (SIZE 2 BYTES),
    parameterLength  INTEGER (SIZE 2 BYTES),
    topicName        NESTEDSTRING (SIZE DEFINED)
}
<transfer>
Back {parameterKind == 5 }
Forward { LENGTH(topicName) == parameterLength }
</transfer>

NESTEDSTRING ::= SEQUENCE {
    nameLength  INTEGER (SIZE 4 BYTES),
    name        OCTET STRING (SIZE CONSTRAINED),
}
<transfer>
    Forward { LENGTH(name) == nameLength }
</transfer>
```

Figure 10. Example of nested Length Constraints.

9. If the network data provided fails a parse, then the packet is ignored and flagged. 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.

4.1. Unique Naming

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, and user-defined types.

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 8 shows the translation from the definition of a DATARSUB sequence in Figure 7. 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 8. 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 4.5 pertaining to the source file generation.

4.3. 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 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.

4.4. 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. It also ensures that a worst-case parse is achieved in linear time. 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 that is generated depends on the SCL definition provided by the user for a protocol.

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 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.

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

4.5. Constraint Transformation

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 section 3 to describe SCL constraints.
Figure 11 shows the parser that is generated as a result of the SCL specification for the sequence in figure 7. 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. 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.

4.5.1. Value Constraints. In figure 7 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 11 lines 8 and 9. 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.

4.5.2. Nested Constraints. The `writerEnt` field in Figure 7 shows a nested value constraint for the DATARSUB sequence. As seen in the generated parser in figure 11, a user-defined type requires the parser 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` or overall PDU data.

4.5.3. Endianness. The example used in section 3 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 constraint in figure 7 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 11. 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 18 of Figure 11.

4.5.4. Length Constraints. Figure 10 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 12 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.

5. Evaluation

The parser generator that was created achieves the five goals described in section 1.1 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 and producing C source code, we are able to ensure that parsers are secure and perform at high speeds.

5.1. Correctness

The generated parsers for the IGMP, RTPS, ARP and NTP protocols were tested against captures from our simulated environment described in [18]. Two different versions of the environment using two different RTPS vendors were tested. The parsers were successfully tested against clean captures, and containing packets with structural RTPS errors. The parser was tested as a unit, and also as a part of the complete multipacket constraint engine.

5.2. Performance

We evaluate the generated parser by comparing to the performance of the SCL implementation described by Marquis et al. [19]. The SCL parser is a universal

```
bool parseDATARSUB (DATARSUB_RTPS *datarsub_rtps, PDU *thePDU,
                    char *progname, uint8_t endianness) {
    if (!lengthRemaining (thePDU, 16, progname)) {
        return false;
    }
    datarsub_rtps->inlineqos = NULL;
    datarsub_rtps->serializeddata = NULL;
    datarsub_rtps->kind = get8_e (thePDU, endianness);
    if (!(datarsub_rtps->kind == 21)) {
        return false;
    }
    datarsub_rtps->flags = get8_e (thePDU, endianness);
    endianness = datarsub_rtps->flags & 1;
    datarsub_rtps->nextheader = get16_e (thePDU, endianness);
    datarsub_rtps->extraflags = get16_e (thePDU, endianness);
    datarsub_rtps->qosoffset = get16_e (thePDU, endianness);
    ENTITYID_RTPS readerent;
    if (!parseENTITYID (&readerent, thePDU, progname, BIGENDIAN)) {
        return false;
    }
    datarsub_rtps->readerent = readerent;
    ENTITYID_RTPS writerent;
    if (!parseENTITYID (&writerent, thePDU, progname, BIGENDIAN)) {
        return false;
    }
    datarsub_rtps->writerent = writerent;
    if (!(datarsub_rtps->writerent.key == 4)) {
        return false;
    }
    if (!(datarsub_rtps->writerent.kind == 194)) {
        return false;
    }
    datarsub_rtps->writerseq = get64_e (thePDU, endianness);
    if (datarsub_rtps->flags & 2) {
        datarsub_rtps->inlineqos = (QOSPARM_RTPS *) malloc (sizeof (QOSPARM_RTPS));
        if (datarsub_rtps->inlineqos == NULL) {
            return false;
        }
        if (!parseQOSPARM (datarsub_rtps->inlineqos, thePDU, progname, endianness)) {
            free (datarsub_rtps->inlineqos);
            datarsub_rtps->inlineqos = NULL;
            return false;
        }
    }
    else {
        datarsub_rtps->inlineqos = NULL;
    }
    if (datarsub_rtps->flags & 12) {
        datarsub_rtps->serializeddata = (TOPICS_RTPS *) malloc (sizeof (TOPICS_RTPS));
        if (datarsub_rtps->serializeddata == NULL) {
            return false;
        }
        if (!parseTOPICS (datarsub_rtps->serializeddata, thePDU, progname, endianness)) {
            free (datarsub_rtps->serializeddata);
            datarsub_rtps->serializeddata = NULL;
            return false;
        }
    }
    else {
        datarsub_rtps->serializeddata = NULL;
    }
    return true;
}
```

Figure 11. Generated parser for DATARSUB sequence.

```
        if (!lengthRemaining (thePDU, pidtopicname_rtps->parameterlength, progname)) {
            return false;
        }
        PDU constrainedPDU;
        constrainedPDU.data = thePDU->data;
        constrainedPDU.len = pidtopicname_rtps->parameterlength;
        constrainedPDU.curPos = pos;
        constrainedPDU.remaining = pidtopicname_rtps->parameterlength;
        NESTEDSTRING_RTPS topicname;
        if (!parseNESTEDSTRING (&topicname, &constrainedPDU, progname, endianness)) {
            return false;
        }
```

Figure 12. Example of a nested length constraint.

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 1 and 2 shows the performance differences between the interpretive parser and the generated parser. These were testing using identical packet captures containing RTPS data.

TABLE 1. 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 2. Generated Parser Performance

Capture Size (MB)	Run Time (s)	Bandwidth (Mbit/s)
77	0.2	3080.00
225	0.7	2571.43
962	2.7	2850.37

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 3 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) [20] 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 3. Realistic Parser Performance

Capture Size (MB)	Run Time (s)	Bandwidth (Mbit/s)
1668	11.5	1191.43
3336	22.0	1213.09
6672	46.3	1152.83

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.

6. Related Work

There has been extensive work in the domain of parser generation using domain-specific languages including Spicy [21], Nail [22], ANTLR [23], and Hammer [24].

Spicy is a recently published DPI framework that generates parsers from a specification language. Similar to our work, Spicy also ensures robust error handling, and allows integration with other software through a custom callback mechanism. They also introduce similar parsing constraints that allow for LL(1) optimizations. Protocols can also be layered as they are modular. Spicy allows for the dynamic detection and dissection of protocols through their concept of sinks. They provide support TCP stream reassembly. Figure 13 shows a subset of the IGMP protocol written in Spicy based on the DNS specification distributed with Spicy. Conversions from packet bytes to internal value are done explicitly (line 5), and a switch structure is used to choose between packet types (lines 6 to 12) in this example. There is an optimized way of specifying context dependent parsing choices.

SCL uses a declarative constraint style that is focused more on a high-level extension to the industry standard ASN.1 notation. Sommer et al. [21] seem to suggest that there is some manual intervention in backtracking (see

```
module igmp;
type PDUType = enum {V3Report=34, Query=17,
    V2Report=22, V2Leave=23};
export type PDU = unit {
    ty:     uint8 &convert=PDUType($$);
    switch ( self.ty ) {
        PDUType::V3Report  -> v3report:
            V3Report();
        PDUType::Query     -> query:
            Query();
        ...
    };
};
type V2Leave = unit {
        maxRespTime: uint8;
        checksum:    uint16;
        groupAddr:   uint32;
};
```

Figure 13. IGMP snippit in Spicy

&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. An example of this is the writerEnt.key constraint on line 15 of figure 7. SCL does not yet support TCP stream reassembly, but as been used to parse mutiple single packet messages including those that use separate frame types.

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.

Autumn[25] is a context sensitive parsing API based on 6 operations: call, transform, snapshot, restore, diff and merge. The call and transform operations represent the parsing options, while the other four operations represent state operations that allow the parser to make parsing decisions based on state information. The generation of the internal representation of the input is still a manual process. Similarly, 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.

Hammer is also an example of a library that allows for the creation of LL(k) recursive descent parsers. It is similarly 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 [26]. 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.

7. Conclusion and Future Work

While significant progress has been made in providing an extended version of SCL to generate fast parsers, there are multiple optmizations and additions that can be made to the parsing framework. We have made the framework available on github[1].

The framework in its current state is only able to examine binary data and does not generate parsers for stream-based data. One of our main goals 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.

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.

In it's 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

[1] http://github.com/alishak/TAG

minimize user error and incorrect parser generation.

In this paper, 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 reliable and secure.

References

[1] Wireshark, "Wireshark Security Advisories," accessed 2017-05-29. [Online]. Available: https://www.wireshark.org/security/

[2] M. Corporation, "Wireshark Vulnerability Trends Over Time," accessed 2017-05-29. [Online]. Available: https://www.cvedetails.com/product/8292/Wireshark-Wireshark.html?vendor_id=4861

[3] M. S. Hasan, A. ElShakankiry, T. Dean, and M. 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*, 2016, pp. 623–628. [Online]. Available: https://doi.org/10.1109/PST.2016.7906997

[4] D. C. Plummer, "An Ethernet Address Resolution Protocol," accessed 2017-05-29. [Online]. Available: https://tools.ietf.org/html/rfc826

[5] B. Fenner, H. He, B. Haberman, and S. H, "Internet Group Management Protocol," accessed 2017-05-29. [Online]. Available: https://tools.ietf.org/html/rfc4605

[6] J. Postal, "User Datagram Protocol," accessed 2017-05-29. [Online]. Available: https://tools.ietf.org/html/rfc768

[7] O. Dubuisson, *ASN.1 Communication Between Heterogeneous Systems*. Morgan Kaufmann, 2001. [Online]. Available: https://books.google.ca/books?id=g7RQAAAAMAAJ

[8] ITU, "ASN.1," accessed 2017-05-29. [Online]. Available: http://www.itu.int/itu-t/recommendations/rec.aspx?rec=x.680

[9] S. Marquis, T. R. Dean, and S. 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*, 2005, pp. 155-164. [Online]. Available: http://doi.acm.org/10.1145/1105634.1105646

[10] "OSPF Version 2 Management Information Base," accessed 2017-05-29. [Online]. Available: https://tools.ietf.org/html/rfc4750

[11] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "A Simple Network Management Protocol," accessed 2017-05-29. [Online]. Available: https://tools.ietf.org/html/rfc1157

[12] J. R. Cordy, "The TXL Source Transformation Language," *Sci. Comput. Program.*, vol. 61, no. 3, pp. 190–210, Aug. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2006.04.002

[13] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider, "Source transformation in software engineering using the TXL transformation system," *Information & Software Technology*, vol. 44, no. 13, pp. 827–837, 2002. [Online]. Available: https://doi.org/10.1016/S0950-5849(02)00104-0

[14] T. R. Dean, J. R. Cordy, K. A. Schneider, and A. 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*, 2001, pp. 622–631. [Online]. Available: https://doi.org/10.1109/ICSM.2001.972779

[15] M. AboElFotoh, T. Dean, and R. 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*, 2009, pp. 112–121. [Online]. Available: http://doi.acm.org/10.1145/1723028.1723043

[16] S. Zhang, T. R. Dean, and S. 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*, 2006, pp. 341–344. [Online]. Available: http://doi.acm.org/10.1145/1188966.1189004

[17] D. Mills, J. Martin, J. Burbank, and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification," accessed 2017-05-29. [Online]. Available: https://tools.ietf.org/html/rfc5905

[18] M. S. Hasan, F. Garcia, F. T. Imam, T. R. Dean, M. Zulkernine, and S. P. Leblanc, "A constraint-based intrusion detection system," in *Proceedings of European Conference on Computer Based Systems*, ser. ECBS 2017. New York, NY, USA: ACM, 2017, to appear.

[19] S. Marquis, T. R. Dean, and S. 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*, 2006, pp. 263–274. [Online]. Available: http://doi.acm.org/10.1145/1188966.1188993

[20] RTI, "Context DDS Professional," accessed 2017-05-29. [Online]. Available: https://www.rti.com/products/dds

[21] R. Sommer, J. Amann, and S. 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*, 2016, pp. 558–569. [Online]. Available: http://dl.acm.org/citation.cfm?id=2991100

[22] J. Bangert and N. 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*, 2014, pp. 158–166. [Online]. Available: https://doi.org/10.1109/SPW.2014.31

[23] T. Parr and K. 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*, 2011, pp. 425–436. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993548

[24] M. L. Patterson, "Hammer," accessed 2017-05-29. [Online]. Available: https://github.com/abiggerhammer/hammer

[25] N. Laurent and K. Mens, "Taming context-sensitive languages with principled stateful parsing," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2016. New York, NY, USA: ACM, 2016, pp. 15–27. [Online]. Available: http://doi.acm.org/10.1145/2997364.2997370

[26] O. Group, "Distributed Relational Database Architecture (DRDA) Standard," accessed 2017-05-29. [Online]. Available: https://collaboration.opengroup.org/dbiop/