LL(k) Optimization of a Network Protocol Parser Generator

Kyle Lavorato
Queen's University
Kingston, Ontario, Canada
kyle.lavorato@queensu.ca

Fahim T. Imam
Queen's University
Kingston, Ontario, Canada
fahim.imam@queensu.ca

Thomas R. Dean
Queen's University
Kingston, Ontario, Canada
tom.dean@queensu.ca

ABSTRACT

During the parsing of an input, there are often cases where a non-terminal can be one of multiple possibilities. In order to avoid trial and error backtracking when parsing, we apply lookahead to evaluate k fields in the possibilities. During this lookahead, we check for fields that have restricted values in order to allow us to determine which of the possibilities we are currently trying to parse. We propose an automated system that is able to conduct this k pass LL(k) lookahead analysis for a Network Protocol Parser Generator. Our LL(k) optimization is a two-module system that first conducts a protocol independent lookahead analysis on network protocol descriptions. We define an XML-style lookahead annotation that encapsulates the full typing information from the lookahead analysis. If any parsing decisions are detected to be optimizable by lookahead, we apply this markup block to them to prepare for code generation. The second module is an existing code generator that outputs a parsing program in C to parse the input protocol. We modify this existing code generator to create optimized lookahead parsers using our parse-identify-repeat parsing algorithm from the encapsulated information in our markup block. This optimized system is able to comparatively outclass the previous version, demonstrating significant improvement.

CCS CONCEPTS

• **Software and its engineering** → **Software evolution**; *Source code generation*; *Parsers*; *Software reverse engineering*; • **Security and privacy** → Network security.

KEYWORDS

Optimization, Grammar, Network Protocols, Protocol Parser, Lookahead, Parser Generator, TXL, Intrusion Detection

ACM Reference Format:
Kyle Lavorato, Fahim T. Imam, and Thomas R. Dean. 2019. LL(k) Optimization of a Network Protocol Parser Generator. In *CASCON '19: 29th Annual International Conference on Computer Science and Software Engineering, Nov 4–5, 2019, Toronto, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

CASCON '19, Nov 04–06, 2019, Toronto, ON

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

One of the most natural ways to optimize any parser is to implement lookahead parsing strategies. This allows the parser to determine which non-terminal it is currently parsing [1, 6, 22]. The benefit of this optimization is to avoid backtracking while parsing. Constant backtracking increases the cost resulting in a performance penalty of any parser. The extreme case is when the only field that can differentiate non-terminal types is near the end of the input.

In variable systems such as an Intrusion Detection System (IDS), the user will be the author of the grammar that needs to be parsed. When generating these systems, users should not be expected to be versed in writing optimal grammars. Therefore it is expected that the user creates a grammar where any of the cases of extreme slowdown may exist. Nevertheless, even in the standard case, lookahead has the potential for a performance improvement.

The lookahead process achieves its performance boost by eliminating general backtracking and correcting some of the inefficiencies identified above. The process does so by determining the correct child non-terminal in a grammar decision for the current parse tree. This is accomplished through parsing the value of a field in the child non-terminal and then matching it to a set of unique pre-defined constants in the grammar for each type.

Since the concept of lookahead is classical in parsing, there has been multiple different approaches within the research community. The ANTLR LL(*) Parser [15], Zebu Protocol Parser [2] and Hammer parsing library [19] all are examples of such systems. Although, each of these systems share a similar downfall in that they are general parsing solutions. They can only be generated through a Backus-Naur Form (BNF) grammar, in which case lookahead parsing is always possible due to non-terminal structure. Our parser system deals with network parsing for an IDS, which does not use BNF grammars, but a description of a protocol as written by a network engineer. Even Zebu which is also a network protocol parser, uses a modified BNF since it only functions for HTTP-like protocols.

In the case of binary protocols, the specification must be analyzed for lookahead opportunities. Here lookahead can only occur if the structure of the protocol allows half-parse identification through value restrictions. The previous work by ElShakankiry et al. [4] implements an LL(1) lookahead for binary protocols, which while useful, is limited in its abilities. Being LL(1), the approach only checks *one* field of each non-terminal, which is not sufficient to differentiate types in many situations. In parsing theory, Parr et al. encounter the same issue with LL(1) parsers and propose that parsers need $k > 1$ lookahead [18].

With overall internet data transfer speeds rapidly rising to over one gigabit per second with the dawn of commercial fibre optic networking [11], there is a need to improve parsing capabilities to process incoming data. Transfer speeds over local Ethernet can

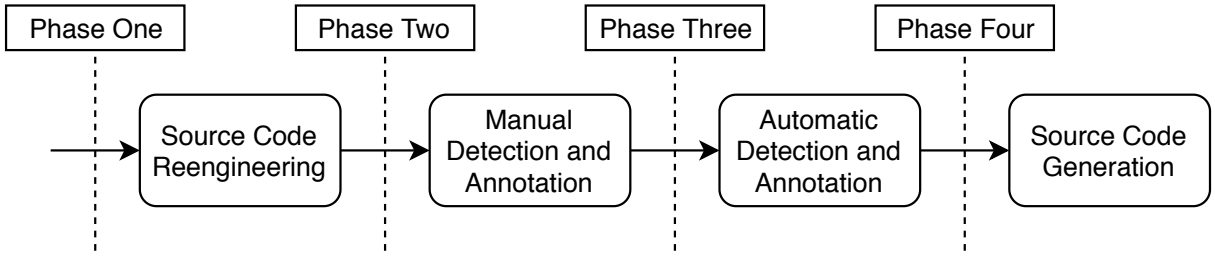

Figure 1: The LL(k) optimization project phases

have even harsher requirements for parsing. The active Ethernet IEEE Standards of 802.3bz [9] and 802.3cb [10] from 2016 and 2018 respectively, specify speeds of a minimum of 2.5 Gbps, up to a maximum of 5.0 Gbps. Parser systems on these local networks must then be able to process at the minimum speed otherwise they will lose input data. When an IDS operates on these high-bandwidth networks using an inadequate parser, it causes large adverse effects on the detection capabilities. If the parser cannot process at the network speed, a queue of input data continues to build until it flushes the entire backlog. When this occurs, the flushed input is allowed through the IDS without any analysis which creates an opportunity for malicious data to bypass the IDS and infect the network. Therefore, optimizing the parser increases the maximum bandwidth it can handle, reducing the potential of this latter scenario to occur.

Figure 1 describes the four phases of our work, with the initial effort containing manual protocol specific analysis and the latter portion entailing the construction of the automatic, protocol independent parsing. A manual lookahead analysis of the Real-Time Publish-Subscribe Protocol (RTPS) and the reengineering of its current backtracking parser into a LL(k) parser is discussed in Section 3.1 and Section 3.2. We discuss the design of an annotation that will model the LL(k) parsing algorithm in Section 3.3. The newly created automated protocol independent lookahead system is discussed in Section 3.4. The automated code generation of the lookahead parser is then presented in Section 3.5.

We validate our approach using a set of standard performance metrics and parsing criteria. From previous optimization work on the parser generator by ElShakankiry et al. [4] we set the metric that performance must increase by at least 6% based on the past comparable modifications using the same data set. The optimized lookahead parser must also satisfy the qualities of adequate parsing for real-time capabilities and system reliability. The proposed optimization is evaluated in Section 4.

2 BACKGROUND

2.1 Network Protocol Parser Generator

The focus of this work is to optimize an existing Network Protocol Parser Generator by ElShakankiry et al. [4], used in the Intrusion Detection System (IDS) by Hasan et al. [5, 20]. The system generates parsers as a set of functions, in the *C* language for binary network protocols. The protocols are described in a custom specification language called SCL [12]. Each SCL file will describe a single protocol and will be individually used as an input for the parser generation. The entire generator is written in the TXL[3] source transformation language, as a pipeline of multiple TXL scripts. TXL pattern matching is used for several annotation processes on the original SCL protocol description, including unique naming for declarations and references and optimization markup. Once annotated, pattern matching is again used for a one-to-one transformation of SCL descriptions of data structures into corresponding C structs for a header file. Following that there is a new transformation from the same annotated SCL file with a one-to-one transformation of the data structures into the functions to parse them. Supporting helper functions are also generated as necessary. The final output of the generator is then a set of header and source files, with one file dedicated to each input specification.

2.2 The TXL Source Transformation Language

TXL is a language designed for source to source transformation and rapid prototyping in software systems [3]. It has evolved over time to become an industry standard general purpose source transformation tool, suited to a wide array of software reengineering tasks. A TXL source transformation is a twofold process. First a context-free grammar set is defined for each of the source languages. The TXL engine then derives a parser from the defined grammar and uses it to process the input in a set of by-example source transformations. The rules and functions in TXL are written to match the contextual information of the input language using grammar elements and transform the input to a specified target language.

2.3 Structure and Context-Sensitive Language

SCL [12] is a language extension to ASN.1 that provides a set of useful mechanisms to define binary network protocols. ASN.1 is a formal notation used for describing data transmitted by telecommunications protocols, regardless of language implementation and physical representation of the data [8]. SCL extends the ASN.1 notation by adding XML style markup and additional data types. SCL was designed based on ASN.1 to provide a familiar notation for Network Engineers to specify their protocols.

An SCL protocol description consists of three main elements: *a program header*, *type decisions* and *rule definitions*. Listing 1 provides an SCL specification example of a Simple Payment Protocol (SPP) for the payment system of a retailer. The module begins with the program header in line 1, which defines the protocol name. Each order of the payment system is represented as the root Protocol Data Unit (PDU) type defined on lines 3-6, containing the `ORDER_HEADER` and the `PAYMENT`. The order header on lines 8-15 is an example of a

```
1 SPP DEFINITIONS ::= BEGIN
2
3 PDU ::= SEQUENCE {
4     header ORDER_HEADER (SIZE DEFINED),
5     paid   PAYMENT (SIZE DEFINED)
6 } (ENCODED BY CUSTOM)
7
8 ORDER_HEADER ::= SEQUENCE {
9     cost      INTEGER (SIZE 2 BYTES),
10    itemCount INTEGER (SIZE 1 BYTES),
11    items     SET OF INTEGER (SIZE CONSTRAINED)
12 }
13 <transfer>
14 Forward { CARDINALITY(items) == itemCount }
15 </transfer>
16
17 PAYMENT ::= (CREDIT | GIFT_CARD)
18
19 CREDIT ::= SEQUENCE {
20     cardNum     OCTET STRING (SIZE 4 BYTES),
21     secCode     INTEGER (SIZE 1 BYTES),
22     billingAdd  OCTET STRING (SIZE 8 BYTES),
23 }
24
25 GIFT_CARD ::= {
26     cardNum     OCTET STRING (SIZE 4 BYTES),
27     cardPin     INTEGER (SIZE 1 BYTE)
28 }
29
30 END
```

Listing 1: An SCL-based protocol specification example

rule definition that describes the contents of the order. It also has a constraint on the length of the list of items on line 14, where the cardinality must be the specified value. Finally, the payment type on line 17 is a type decision which specifies that the *paid* field of the PDU is an opaque container that can hold one of the two rule definitions of `CREDIT` or `GIFT_CARD`. It is then up to an external system to determine which type is inside the instance of *paid* in the PDU to decode the payment information of the order.

The SCL `transfer` block serves the most significant role both for the original parser developed by ElShakankiry et al. as well as our optimization. The transfer markup specifies the relevant constraints when decoding the data through parsing. The two transfer statements that the parser must consider when decoding any packet are specified using `Back` and `Forward` blocks in SCL. They both represent a conditional statement that must hold true during a parse. Otherwise, the packet is determined to be malformed and must fail the parse. Using the `Back` constraint, the condition is evaluated once the decoding of the data is complete. Similarly, in a `Forward` constraint, the condition is evaluated before or during the decoding of the data. Our primary concern is the `Back` constraints as they allow us to look ahead using values that must hold true for a parse to succeed.

2.4 The Real-Time Publish-Subscribe Protocol

The Real-Time Publish-Subscribe (RTPS) protocol [14] is designed by the Object Management Group (OMG) for unidirectional data exchange on multicast systems. It is a protocol where a set of applications publish data to the local caches of subscribers. A standard RTPS message is defined in Figure 2, which is composed of a header and a set of submessage elements in a type decision scenario. There are nine different types of submessage in RTPS, representing the main functionality of the protocol. For this paper, we will consider a simplified version of the RTPS protocol with four types of submessages in the type decision. This simplified version consists of the `Heartbeat`, `Data`, `Data(P)` and `Data(R)` submessages as defined by the protocol. We have excluded the `Info_dst`, `Info_ts`, `Acknack` and `Gap` submessages as they are structurally similar to the heartbeat message. The `Data(W)` is also excluded due to its similarity to the `Data(P)` and `Data(R)`.

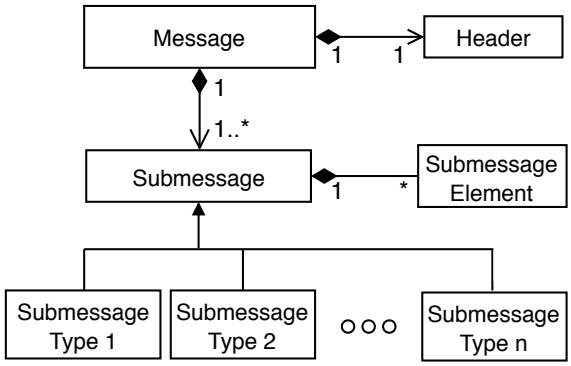

Figure 2: The RTPS PDU structure

The submessage level definition of the simplified RTPS protocol in SCL is defined in Listing 2. The `SUBMESSAGE` element in the RTPS PDU of Figure 2 is specified by the SCL type decision definition on lines 1-2 where it can be one of the four types. Each submessage sub-type in the remainder of the figure has its own rule definition that defines it and any constraints on its data fields. Each of these submessage rule definitions also have one or more back statements that place a value restriction on fields, which can be used in lookahead analysis.

3 OPTIMIZING THE PROTOCOL PARSER

We conduct a brief packet analysis on a capture from a simulated Air Traffic Control network implemented using RTPS. The packet capture represents the standard network traffic our IDS experiences, where it was revealed that out of the total 677, 326 packets, 98.08% were RTPS packets. Additionally, in those packets, there were 1, 388, 354 `SUBMESSAGE` type decisions. It was also discovered that 9, 654, 268 backtracks were required to parse all of those `SUBMESSAGE` elements using the original parser by ElShakankiry et al. This analysis indicates a clear inefficiency with the parsing of `RTPS SUBMESSAGE` type decisions in the original parser and led us to develop the optimization approach presented in this paper.

```
1  SUBMESSAGE ::= (HEARTBEAT | DATA | DATA(P)
2                 | DATA(R))
3
4  HEARTBEAT ::= SEQUENCE {
5      kind        INTEGER (SIZE 1 BYTES),
6      flags       INTEGER (SIZE 1 BYTES),
7      nextHeader INTEGER (SIZE 2 BYTES),
8      readerEnt   ENTITYID (SIZE DEFINED) BIGENDIAN,
9      writerEnt   ENTITYID (SIZE DEFINED) BIGENDIAN,
10     firstSeq    OCTET STRING (SIZE 8 BYTES),
11     lastSeq     OCTET STRING (SIZE 8 BYTES),
12     count       INTEGER (SIZE 4 BYTES)
13 } (ENCODED BY CUSTOM)
14 <transfer>
15 Back {kind == 7}
16 </transfer>
17
18 DATA ::= SEQUENCE {
19     kind        INTEGER (SIZE 1 BYTES),
20     flags       INTEGER (SIZE 1 BYTES),
21     nextHeader INTEGER (SIZE 2 BYTES),
22     extraFlags INTEGER (SIZE 2 BYTES) POS,
23     qosOffset   INTEGER (SIZE 2 BYTES),
24     readerEnt   ENTITYID (SIZE DEFINED) BIGENDIAN,
25     writerEnt   ENTITYID (SIZE DEFINED) BIGENDIAN,
26     writerSEQ   INTEGER (SIZE 8 BYTES),
27     inlineQos   QOSPARM (SIZE DEFINED) OPTIONAL,
28     serializedData OCTET STRING (SIZE CONSTRAINED)
29 } (ENCODED BY CUSTOM)
30 <transfer>
31 Back {kind == 21}
32 Back {writerEnt.kind == 2 || writerEnt.kind == 3}
33 </transfer>
34
35 DATA(P) ::= SEQUENCE {
36     kind        INTEGER (SIZE 1 BYTES),
37     flags       INTEGER (SIZE 1 BYTES),
38     nextHeader INTEGER (SIZE 2 BYTES),
39     extraFlags INTEGER (SIZE 2 BYTES),
40     qosOffset   INTEGER (SIZE 2 BYTES),
41     readerEnt   ENTITYID (SIZE DEFINED) BIGENDIAN,
42     writerEnt   ENTITYID (SIZE DEFINED) BIGENDIAN,
43     writerSEQ   INTEGER (SIZE 8 BYTES),
44     inlineQos   QOSPARM (SIZE DEFINED) OPTIONAL,
45     serializedData PARTICIPANTS (SIZE DEFINED)
46 } (ENCODED BY CUSTOM)
47 <transfer>
48 Back {kind == 21}
49 Back {writerEnt.kind == 194}
50 Back {writerEnt.key == 256}
51 </transfer>
52
53 DATA(R) ::= SEQUENCE {
54     kind        INTEGER (SIZE 1 BYTES),
55     flags       INTEGER (SIZE 1 BYTES),
56     nextHeader INTEGER (SIZE 2 BYTES),
57     extraFlags INTEGER (SIZE 2 BYTES),
58     qosOffset   INTEGER (SIZE 2 BYTES),
59     readerEnt   ENTITYID (SIZE DEFINED) BIGENDIAN,
60     writerEnt   ENTITYID (SIZE DEFINED) BIGENDIAN,
61     writerSEQ   INTEGER (SIZE 8 BYTES),
62     inlineQos   QOSPARM (SIZE DEFINED) OPTIONAL,
63     serializedData TOPICS (SIZE DEFINED) OPTIONAL
64 } (ENCODED BY CUSTOM)
65 <transfer>
66 Back {kind == 21}
67 Back {writerEnt.kind == 194}
68 Back {writerEnt.key == 4}
69 </transfer>
```

Listing 2: Four RTPS submessage structures in SCL

3.1 The Lookahead Mechanism

Since the type decision mechanism using the one pass LL(1) in the original parser was not efficient and effective for RTPS SUBMESSAGE types, we conducted a manual k pass lookahead analysis to determine the optimization possibilities. We examined the SCL definition of the SUBMESSAGE type decision as defined in Listing 2. The Back statements of each type definition in the SCL specification are used to set the restricted values for different fields. For example, the statement Back {kind == 21} in line 48 for the DATA(P) definition indicates that the value of the kind field must be 21.

The HEARTBEAT type can be identified by the existing single pass lookahead as it has a unique value of 7 for the kind field in the SUBMESSAGE type decision. However, DATA, DATA(P) and DATA(R) share the same value restriction of 21 on the kind field. Therefore, the LL(1) approach would be unable to determine a submessage's type from the first restriction; i.e., when the value 21 is parsed it could be any one of three possible types. The lookahead process must then continue past pass one for these three data types.

Examining the value constraints on the three remaining data types in Listing 2, they all share a restriction on the writerEnt.kind field. In addition, the intermediate fields of flags, nextHeader and readerEnt create an equal byte offset of 7 bytes across all types. The DATA type is then uniquely identified by a restricted value of either 2 or 3 of the writerEnt kind field, completing its lookahead. The DATA(P) and DATA(R) types once again have a non-unique value restriction of 194 on that field, requiring another pass of lookahead to possibly differentiate the two.

The final pass of the lookahead examines the remaining Back constraint on the writerEnt.key field. Both DATA(P) and DATA(R) can then be uniquely identified through restricted values of 256 and 4, respectively. Additionally, since the writerEnt field is already common with the last pass, the lookahead can proceed without additional parsing as the byte offset would be null.

The results of our manual lookahead analysis indicate that a lookahead parser can be generated for the SUBMESSAGE structure based on its value constraints uniquely identifying each message type. Expanding the same analysis to include the entire RTPS protocol suggested no additional type decisions in RTPS that could be optimized by further lookahead when the existing LL(1) fails. Based on the conducted analysis, we determine that optimizing type decisions by looakahead analysis has three requirements:

(1) There must be some field v_k, where k is the current lookahead pass, that has a restriction on its value for all types in the type decision.
(2) The byte offset, or total size of all fields in any type in the type decision from either the beginning or the previous v_{k-1} value, to v_k are equal to locate the data in the PDU across all types.
(3) The value of v_k must be unique across all types in the type decision or there must be a v_{k+1} that can be evaluated in a subsequent pass. With conflicting v_k values, the type cannot be resolved. If no v_{k+1} is found, the type decision cannot be optimized by lookahead.

The automated LL(k) annotation process that we have implemented realizes each of the requirements above during its analysis phase.

3.2 The LL(k) Parsing Algorithm

We define the general case of our switch-parsing algorithm for lookahead on type decisions in Listing 3. It runs on a *parse-identify-repeat* concept for the fields with restricted values on them. We first parse all fields indicated by the byte offset, including v_k, to retrieve the restricted value instance. We then perform a switch statement based on the result of v_k as shown in line 3.

If v_k represents a unique value for any type then we have a case statement on that value. There are two possible behaviour sets inside any case statement. The first instance is when a unique value fully defines a type as with `Unique1Val` on lines 4-7. Inside the case block, we then call a newly defined version of the function to parse that specific type identified by appending ¨_0¨ to its name. This optimized version passes any fields that were already parsed as part of the byte offset as parameters in addition to the standard function parameters. Since the parsed fields are passed onward, no effort is wasted re-parsing them in the typed parse function. After parsing we save the data and the identifier for the type we parsed in our result when successful.

If v_k instead represents a conflicting case value, the second type of case block instance will begin the next pass of the lookahead as seen in lines 8-21. The process repeats for v_{k+1}, retrieving and switching until we reach the final v_n, where n is the total number of passes. At any point when the type decision is typed and parsed, we exit the switch statement and return true to mark the parse as successful. If typing is unsuccessful the switch will fall through to the default case on lines 22-23 to mark the parse as unsuccessful. This is to ensure that a malformed or malicious packet which may possess a non-standard value for that protocol still fails the parse. This new *parse-identify-repeat* lookahead algorithm replaces the existing inefficient backtracking algorithm in our original parser.

Listing 4 presents the traditional backtracking algorithm. The algorithm first tries to parse the type decision as one type (Lines 2-6) and if that fails, it reverts to the initial parse point and tries a new type (Lines 7-11). This process continues until the algorithm succeeds parsing a type or attempts all types without a success. Significant effort is therefore wasted in trying to parse incorrect types using the traditional backtracking approach. When the byte offsets are large, the algorithm can parse the majority of a message before determining it as the wrong type. All parsed work and retrieved values are therefore lost when backtracking and must be reparsed on the next attempt. Examples of such are the cases of parsing RTPS `DATA(P)` and `DATA(R)` messages.

3.3 Grammar Extension

To generate the source code for the LL(k) parser, we first needed to annotate the SCL description of a protocol with a model of the desired output code. We therefore proposed an XML style extension to the SCL grammar that would accomplish the task of annotation. In order to function as a robust solution, three requirements were considered: it must encapsulate all the necessary details for code generation; be able to be generated automatically from the initial SCL description and; and finally, seamlessly integrate into the SCL grammar. Figure 3 describes the addition to the SCL grammar, where the base `lookahead_block` would be included sequentially after a transfer block. Since the lookahead block is used in an automatic

```
1 bool parseTYPEDECISION(RESULT * result)::
2     FirstUnique = ParseField()
3     switch (FirstUnique)
4     case Unique1Val :
5         ... Parse Type1_0 ...
6         result->type = TYPE1_VAL
7         break
8     case Unique23Val :
9         FieldTwo = ParseField()
10        SecondUnique = ParseDefinedStruct()
11        switch (SecondUnique->Entry1)
12        case Unique2Val:
13            ... Parse Type2_0 ...
14            result->type = TYPE2_VAL
15            break
16        case Unique3Val:
17            ... Parse Type3_0 ...
18            result->type = TYPE3_VAL
19            break
20        default:
21            return false
22    default :
23        return false
24    return true
```

Listing 3: Optimized Lookahead algorithm

```
1 bool parseTYPEDECISION(RESULT * result)::
2   position = PDU->curPos
3   remaining = PDU->remaining
4   if ( ... Parse Type1 ... )
5     result->type = TYPE1_VAL
6     return true
7   PDU->curPos = position
8   PDU->remaining = remaining
9   if ( ... Parse Type2 ... )
10    result->type = TYPE2_VAL
11    return true
```

Listing 4: Unoptimized backtracking algorithm

TXL transformation into parser source code, each element will be a one-to-one mapping from the block to source code to handle the detail encapsulation. This new annotation block is then generated on type decision rules.

An example of the full lookahead block that models the new parsing for our simplified RTPS protocol is shown in Figure 5. The entirety of the lookahead block is contained inside an XML style `<lookahead>` header and footer. Each `pass_level` is a set of statements representing another level of lookahead can be seen grouped in lines 2-4, 5-10, and 11-14 respectively. These blocks then map to the parsing algorithm's switch statements. The initial `@[id]` on the first line of each set represents the switching parameter, which is the field with a restricted value that each type of the type decision contains. Each of the following `switch_case` statements represent one case statement with the type's unique value of the

⟨*lookahead_block*⟩ ::= <lookahead> {⟨*pass_level*⟩} </lookahead>

⟨*pass_level*⟩ ::= @ ⟨*id*⟩ {⟨*switch_case*⟩} END [⟨*req_parse*⟩] {⟨*new_level*⟩}

⟨*switch_case*⟩ ::= ⟨*id*⟩ @ ⟨*number*⟩

⟨*req_parse*⟩ ::= <⟨*parse_item*⟩ {, ⟨*parse_item*⟩}>

⟨*parse_item*⟩ ::= ⟨*id*⟩ @ ⟨*parse_size*⟩

⟨*parse_size*⟩ ::= ⟨*number*⟩
| ⟨*id*⟩

⟨*new_level*⟩ ::= ⟨*id*⟩ {, ⟨*id*⟩} @ ⟨*number*⟩

Figure 3: Grammar for the lookahead block

```
1 <lookahead>
2    @ kind
3        HEARTBEAT @ 7
4    END <kind @ 1> DATA(P), DATA, DATA(R) @ 21
5    @ writerEnt.kind
6        DATA @ 2
7        DATA @ 3
8    END <flags @ 1, nextHeader @ 2, extraFlags
9    @ 2, qosOffset @ 2, readerEnt @ ENTITYID,
10   writerEnt @ ENTITYID > DATA(P), DATA(R) @ 194
11   @ writerEnt.key
12       DATA(P) @ 256
13       DATA(R) @ 4
14   END
15 </lookahead>
```

Listing 5: Full lookahead block checking past LL(1)

restricted field. The `END` represents the final line of the current `pass_level` and contains the closing information for the next one.

The `req_parse` is an optional element inside the `< >` set that contains all the fields and their corresponding size, or the types that are part of the byte offset, including the switching parameter. These must then have their value retrieved before the switch case can be evaluated. This translates to statements that declare variables of the identified size or type for the fields. They then retrieve the correct number of bytes sequentially from the packet, parsing the fields. In the case of the first pass in line 4 of Listing 5, only the `kind` field must be retrieved before evaluating a switch statement on `kind`. In the second pass on lines 8-10 several fields must be parsed before the switch statement can be evaluated on `writerEnt`. Finally, the `new_level` in Figure 3 is a set of statements that show all the types that did not have unique restricted values. If not empty, these types would become the subject of the next level of the lookahead in order to check if there is anything further that can differentiate them. This corresponds to the case statement holding the next switch statement described in the next `pass_level`. In line 4 of the example in Listing 5, `DATA(P)`, `DATA` and `DATA(R)` must have a value of 21 for kind; i.e., they cannot be differentiated in the initial switch statement. Therefore, the case for the value 21 would hold a new switch statement to differentiate the two through a new restricted field.

3.4 Automatic SCL Analysis

After developing the lookahead annotation mechanism, we developed an automated approach to analyze any protocol description and search for scenarios where lookahead parsing can be applied. A TXL script was written to run against the SCL protocol descriptions before code generation. Our approach provides a robust solution that can perform k pass lookahead analysis on any SCL input and produce an output SCL file where `lookahead_block` elements are annotated to any optimizable type decisions. The approach is fully automated and therefore requires no human interventions.

Lookahead Pass One. The k pass lookahead system functions off of pattern matching in the TXL language. This allows us to define general case patterns in the SCL grammar that correspond to the information related to lookahead. We begin the first pass of the lookahead by taking advantage of the existing one pass lookahead in the existing parser generator by ElShakankiry et al. This LL(1) system examines only the first restricted value field of types in a type decision. To determine if it can identify types based on the first field, it annotates the type decision in the following form:

```
FULLID ^ SHORTID [@ optimizable] ::= (TYPE1 @ SIZE VALUE
| TYPE2 @ SIZE VALUE).
```

When annotation of the type decision is complete, it checks if all the annotated values are unique. If that is the case, the `@ optimizable` tag is added to notify the code generator to generate a single field lookahead parse. While this method is functional, it creates a messy and hard to understand annotation. Therefore, our lookahead analysis only attempts lookahead on any type decisions that do not have the *@ optimizable* tag to save processing time. On any type decisions that do not have the tag, we retrieve the unique value information directly from the existing annotation. Any missing information, such as the field name can be discovered from the rules directly. It is then used in the construction of the first level of the lookahead block.

Arbitrary Recursive Lookahead. We define a recursive algorithm for the remaining passes of the lookahead. This involves a three step analysis process to determine if any restricted value field allows for a type to be identified in a type decision. In the first step, we analyze the SCL back statements in the definitions of one type in the type decision for a new common restricted value field. In the remaining types, we check the k^{th} restricted value, which must be on the same field across all types to satisfy requirement one of lookahead. This corresponds to detecting the field and value from the k^{th} back statement of the pattern `Back {ID == VALUE}`. If the ID matches for all types, we proceed to the next step.

The second step is to examine the intermediate fields in each of the types to meet the second requirement of the lookahead process. Past pass one, we would always have intermediate values to examine to ensure an equivalent byte offset. We first retrieve the previous position in the types from the `req_parse` of the previous pass and then select one type definition to sample. We collect the field names and sizes of all items between the previous switching

parameter and the new restricted field. This information is then recorded in a list and compared to the definition of each of the other remaining types in the type decision. As long as every comparison returns a true result, we proceed to step three.

The third and final step is to determine which types can now be uniquely identified, to complete the third requirement of lookahead. The k^{th} back statement in each type is evaluated again to examine the restricted value. Each encountered value is stored in a used value list, along with the type it belongs to. Then when the next type is evaluated, its restricted value is first compared to the used list to see if it is already present. If so, it is added to a conflict list instead as it is not unique. Once all remaining types have been analyzed, the conflicting values already in the used list will be removed and added to the conflict list instead. This information is then passed forward to the annotation phase of the current pass.

This recursive descent process continues until one of two conditions are met. The first termination condition is triggered when there is an empty conflict list at the end of any pass of the algorithm. This indicates successful identification of each type in the type decision from a subset of its restricted value fields. At this time the algorithm appends the constructed lookahead block to the type decision in the SCL for future code generation. The other termination condition is a failure fall-through when the algorithm fails to meet the requirements for any of the three steps. This indicates that the algorithm has failed to find a subset of predefined values that can support a lookahead type decision. At this point, the current progress of the lookahead block is scrapped, as we only ever append a complete lookahead block to the type decision.

Although k is defined as unbounded for our algorithm, one of the two exit conditions would always trigger, causing the algorithm to terminate. The unbounded k allows the algorithm to find an optimal parse through lookahead when possible, as it can check an unlimited number of back statements. In theory, an unbounded k could potentially have the algorithm unsuccessfully searching for a differentiating value set for a long time. In practice, protocols traditionally have a limited number of fields with required or identifying values, assigning a finite value to k. Additionally, since this algorithm runs only once during the setup of a new system, a costly analysis phase can only ever slow down the build time, which is not a concern in the requirements of the system.

3.5 Parser Code Generation

The parser generation process with lookahead is depicted in Figure 4. We have introduced a new pattern of a type decision with a lookahead block, where previously type decisions did not have any additional blocks. Therefore, we must add a new TXL rule to the current generator to handle the pattern in addition to the current type decision transformation rule for standard backtracking type decisions. This new transformation rule follows the mapping of the lookahead block to source code and creates the lookahead function that parses the type decision element. As part of the code generation, supporting functions to parse the types must also be generated. We define another TXL rule to create the optimized function to parse the types inside the type decision and append an ¨_O¨ to the name, as specified in Section 3.2. In the case of the `HEARTBEAT` type the function declaration would be of the following

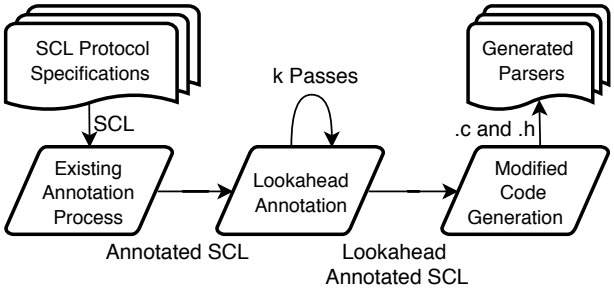

Figure 4: The parser generation process

form that contains all old parameters including the parsed *kind* field:

```
bool parseHEARTBEAT_O (HEARTBEAT * heartbeat, PDU *
thePDU, char * progname, uint8_t * kind, uint8_t
endianness);
```

The lookahead analysis and code generation of the system must only be completed a single time under standard use. Additionally, the lookahead analysis in our system is completed before code generation instead of occurring as packets arrive during runtime. This allows the system to generate an optimized parser at no effective cost to the runtime bandwidth.

4 EVALUATION

We evaluate the success of our proposed k pass lookahead optimization by comparing the generated system for the RTPS parser by the base version generated by ElShakankiry et al. [4].

Real Time Parsing. To correctly assess the *Real Time Parsing* requirement of the optimization we first measured the performance of the systems. Testing was performed by running both parsing algorithms in isolation, using a 1.678 GB packet capture of network data. The sample is generated using the RTI version of the Data Distribution Service (DDS) [21] and is similar to the traffic the parser would receive under normal use, containing RTPS, IGMP, ARP and NTP packets.

Table 1: Specifications for the test hardware platforms

System	CPU Frequency	Memory
Raspberry Pi 2	4 Core–0.90 GHz	1 GB
Low Tier	2 Core–3.00 GHz	3.5 GB
Mid Tier	4 Core– 3.00 GHz	16 GB
High Tier	6 Core 4.30–GHz	32 GB

To determine the scalability of the optimization across all systems, we test the parsing algorithms on varying hardware. This includes a range of low-end to high-end consumer hardware specified in Table 1, using Linux Mint 19. A full set of 200 parses were performed for both parsers to measure the bandwidth of the system, as a measure of their ability to parse data. An average value was then calculated from the 200 trials to negate the possibility of varying system load affecting the measurements.

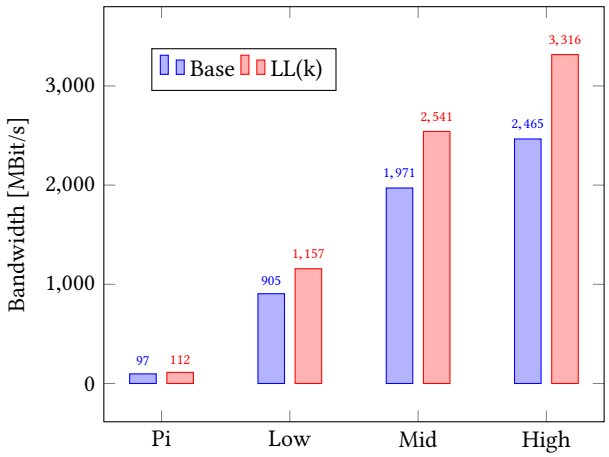

Figure 5: Comparison of parsing capabilities

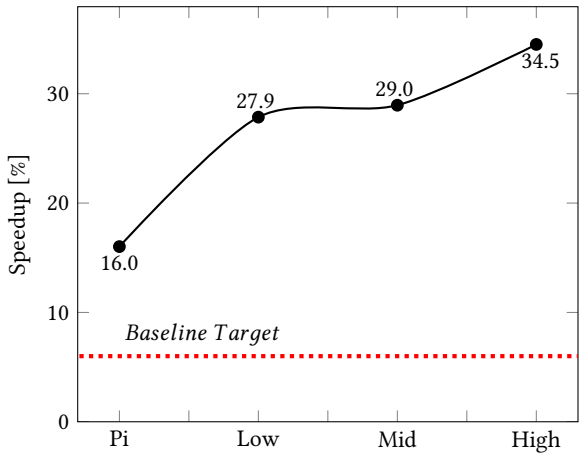

Figure 6: Relative speedup of the lookahead algorithm

Figure 5 demonstrates the performance differences between the parsing algorithms on each platform. Across all hardware, the lookahead parser provides a performance increase over the backtracking parse algorithm. Figure 6 depicts the amount of relative speedup between the base and lookahead algorithms, as well as our minimum speedup metric as a baseline. From the chart it can be seen that the performance value is dependant on the hardware, with the lowest end achieving a *16.02%* increase and the high end achieving *34.51%*. Overall, the relative speedup is not only positive in all cases but it increases steadily with each upgrade in hardware.

Considering the performance results, we observe a clear speedup with the LL(k) optimization over the non-optimized version. Even the lowest speedup of 16.02% surpasses the real time parsing requirement of a 6% increase in bandwidth, while the best increase of 34.51% completely outperforms it. Additionally, where the base system was not able to reach the required 2.5 Gbps of the current Ethernet standard, the lookahead parser reaches those speeds even on mid-tier hardware. These results indicate that our optimization approach is successful in improving the real-time parsing capabilities of the parser.

Additionally, since parsers are traditionally bottlenecked by the hardware they operate on, the scalability with the better hardware indicates further success. In each case where the hardware is improved, the base system is able to achieve a higher bandwidth, independent of any other changes. In accordance with this higher computing power, the lookahead algorithm is able to gain an even larger relative speedup than the previous hardware tier. With this positive relationship, it indicates that the optimization is able to present an even larger net increase of bandwidth on more powerful hardware. For example, the mid-tier hardware presented a net increase of *570.5* MBit/s while the high tier hardware experienced a net increase of *850.7* MBit/s from the same optimization. Therefore, the new lookahead approach does not suffer from any scalability costs and presents a clear trend that allows performance to increase even further with better hardware.

System Reliability. To correctly assess the *Reliability* of the optimized system, we must ensure that its ability to parse packets for all protocols has not been affected. Therefore, to validate the reliability we perform two different tests. We first inspect the parser's ability to successfully parse packets in a traffic sample. We again run both versions of the parser on the same 1.67 GB capture of network data as before, while recording the statistics. To accurately judge reliability, we must both inspect the parse count for RTPS and all other protocols. In this test, the base system correctly parsed $10,707,581$ total packets, out of which $10,518,612$ were RTPS packets. The test is then repeated with the lookahead parser, which achieves the same packet success count. Since the packet parse count remains unchanged, there is no unexpected behaviour causing parses to fail in the generated parses for all protocols.

Another aspect of reliability that we must also evaluate is the details of memory usage by our parsers. For a parser to be reliable it must not only be able to parse packets consistently, but it must be able to run uninterrupted. To validate the parser's ability to execute indefinitely, we must ensure it has no memory leaks during standard operation. We use the Memcheck memory error detector by Valgrind [23] for this evaluation. The tool was run with the system executable to analyze the heap while the parser performed its tasks. For the 1.67 GB packet capture it determines at program termination that 0 bytes remain in 0 blocks on the heap. Additionally, there is a total of 37,079,533 allocations and a matching 37,079,533 frees on the heap with a total of 2.61 GB allocated over the execution lifetime. This ensures that the system will have no issue being run for long periods to parse real-time data.

5 RELATED WORK

Parsing or the decoding of an input is an important topic across all fields of computing, not only in Network Security. Due to its large prevalence and history, several systems have emerged as the top options in parsing technologies. As the problem of parsing theory and efficient parsing is a classic one, there are countless solutions of different ways to create lookahead parsers. We review solutions that also implement lookahead strategies such as ANTLR [15], ANTLR4 [16], Zebu [2] and Hammer [19].

ANTLR LL(*) Parser Generator. The ANTLR LL(k) system is one of the most common parser generator frameworks, designed by

Parr et al. [17] in 1995. The base system is a classical example of a general parser generator. Upon being presented with a context-free grammar, it will create a LL(k) parsing system that can decode any data encoded in that grammar.

Since then, the parser generator has been continually optimized and improved to create a LL(*) parsing strategy, and an associated grammar analysis system [15]. LL(*) is a parsing optimization that throttles up from the conventional $k \geq 1$ to an arbitrary lookahead. A grammar augmented with syntactic and semantic predicates and embedded actions is used as the input to the ANTLR generator. Their syntactic predicates allow for the arbitrary lookahead of the LL(*) decision. They are implemented as grammar fragments that must match the desired input, much like the XML style annotations that we use in our SCL grammar.

ANTLR4 is an improved version that enhances the work of the previous LL(*) parse, introducing the ALL(*) parsing strategy [16]. In the ALL(*) strategy, they again progress the previous method by moving the grammar analysis to parse time. This allows ALL(*) to create a LL(*) decision set for all the non-left recursive grammars, where LL(*) on its own only worked efficiently with LL-regular grammars.

Although the arbitrary lookahead of LL(*) is a more efficient approach, lookahead opportunities must be developed by analyzing the grammar only, due to the general nature of ANTLR. This is a different approach from our proposed LL(k) strategy as its lookahead is dependent on semantic constraints placed on the data structure in the input specification. As a result, our arbitrary lookahead will not be able to cease optimization attempts when the lookahead decision is too complex. It will always iterate through every possibility for lookahead, but due to the nature of our constraints, the upper bound will rarely exceed $k \leq 5$. With regards to the ALL(*) parsing strategy, we employ a similar optimization, as we perform our lookahead analysis while generating the parser. As a result, the generated parser does not have to deal with any lookahead checking as it parses its input.

ANTLR is a general-purpose parsing framework which cannot be easily compared to our system. Its general case nature requires a BNF grammar for the source to be parsed. Our generator respectively uses a specification in SCL of the network protocol to be parsed due to its IDS centered design. While it is possible to convert a protocol description to BNF, it is outside the scope of this research. However, we make available the input packet captures[1] used in our evaluation for future studies.

Zebu Network Protocol Parsing. Zebu is a domain-specific language that is an annotated version of Augmented BNF (ABNF), used to generate network protocol parsers for HTTP-like protocols. It uses a unique lookahead strategy to optimize their parsing. They implement a two-layer parsing tactic with a coarse-grained generic parser and multiple dedicated parsers. The generic parser acts as their lookahead, searching for high-level message elements in a packet while ignoring everything else. Once a high-level element such as a header is found, the generic parser is paused and a dedicated parser for that particular element is executed. After its completion, the generic parser resumes its lookahead.

[1] http://pyxis.ece.queensu.ca/data/parser/

Zebu uses a standard input format of ABNF with only slight modifications, whereas our system requires an annotated version of the industry standard ASN.1. This allows Zebu to be slightly simpler for a first-time developer to pick up and begin using comparatively. Instead of a pattern matching tool like TXL, Zebu annotations are processed by the PCRE library [7]. They are processed as an input of regular expressions and then transformed into stub functions. The use of a library provides a much lower learning curve for other developers to extend the system in the future. In contrast, our work relies on the TXL language for the annotation and code generation. While TXL is a powerful tool, it has a significant learning curve to begin future additions.

Zebu's two-layer lookahead is an interesting strategy but in a different domain as our work. Our goal is to create a zero cost lookahead optimization for network protocols through lookahead before runtime. Zebu comparatively generates a parser that completes its lookahead during the parsing process. This mid-parse lookahead carries the downside of an added cost as it requires additional processing time to resolve types that ours does not. Despite this latter difference, the two-layer lookahead is a tactic that could still be applied to parser generation as a special lookahead routine. In a protocol with complicated semantic constraints, the two-pass system could aid in the detection and verification of an optimizable data structure. This would remove the need to search through every reference of the semantic constraints as we would have dedicated techniques for each constraint.

Hammer. Hammer is an open source *C* library that can be used to create LL(k) recursive descent parsers. It is specifically designed to parse binary protocols, which is only a subset of the protocols that our system must generate parsers for. While Hammer also applies the LL(k) optimization at generation time like our proposed optimization, it has little to no documentation despite its open source nature. Therefore the actual optimization methodology cannot be determined without reengineering their source code.

6 FUTURE WORK

While our optimization in its current state has successfully accomplished its goal of a significant performance increase, there are multiple extensions that will allow the optimization to be relevant in additional scenarios and provide further performance benefits.

Partial Lookahead Parsing. There are cases where perhaps only half a type decision can be uniquely typed by k levels of lookahead. Our optimization is currently an all-or-nothing approach that will revert to a backtracking parse if the entire type decision cannot be uniquely identified. We propose that the system be extended to generate the lookahead case statement parse for any structures that can be typed with lookahead. The type decision can then be completed by generating a backtracking parse in the conflicting case statement for the remaining types. This will eliminate a non-trivial amount of backtracking in most cases, raising the overall effectiveness and reach of this optimization. This change will only require slight modifications to the annotation and code generation. The removal of the annotation if the all-or-nothing process fails must be adjusted to keep the current annotation progress as-is. Then the code generation must have a slight adjustment to where if it does

not find an empty <new_level> at then end of the annotation then it must generate the backtracking parse for those remaining cases.

Evaluation of Complex Constraints. The system currently only evaluates simple Back constraints on fields to determine the unique values. In the vast majority of cases this will run unimpeded as protocols traditionally have only specific integer constraints on their fields. Although there are cases in binary protocols such as in SMB2 [13], where we require a complex Back statement in the form of Back{padding == 8 - ((64 + 36 + dialectCount * 2) % 8)}. This is because SMB2 uses variable length padding that must ensure that the next field is 8-byte aligned. Due to TXL requiring exact pattern matching, our lookahead analysis will only match on back statements consisting of a single defined value on the right side of the equality condition. As a result, this could mark a type decision as unoptimizable when in actuality the complex constraint would evaluate to unique values. Although these complex requirements are uncommon across most binary protocols, it would still be beneficial to increase the number of Back constraints patterns that the algorithm can recognize.

7 CONCLUSION

In this paper, we presented an optimization approach to improve an existing one pass lookahead parsing system and extend it to k passes. The two-module system that we have presented is a secure generic solution that is able to conduct a k pass lookahead on any network protocol defined in SCL. We have also defined a generic XML style annotation that can even be used with non-network protocol parser generators for lookahead analysis. Additionally, it is fully automatic requiring zero user input to function. Due to its pattern matching nature it can be easily expanded to handle special cases as they arise through the addition of new TXL rules for specific patterns. The proposed lookahead approach eliminates a large amount of wasted effort from backtracking to improve the bandwidth of the system. As a result, it is able to achieve a relative speedup as high as 34.51% to improve the real-time parsing capabilities of the system, based on our evaluation presented in the paper. The proposed approach also avoids wasted effort by finding lookahead opportunities before parser generation, requiring no additional work during parser execution.

ACKNOWLEDGMENTS

The authors acknowledge the support of the Natural Sciences and Engineering Research Council (NSERC) and the Department of National Defence (DND) for this research.

REFERENCES

[1] Theodore P Baker. 1981. Extending lookahead for LR parsers. *J. Comput. System Sci.* 22, 2 (1981), 243–259.
[2] L. Burgy, L. Reveillere, J. Lawall, and G. Muller. 2011. Zebu: A Language-Based Approach for Network Protocol Message Processing. *IEEE Transactions on Software Engineering* 37, 4 (July 2011), 575–591. https://doi.org/10.1109/TSE.2010.64
[3] James R. Cordy. 2006. The TXL source transformation language. *Science of Computer Programming* 61, 3 (2006), 190 – 210. https://doi.org/10.1016/j.scico.2006.04.002 Special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA âĂŹ04).
[4] A. ElShakankiry and T. Dean. 2017. Context Sensitive and Secure Parser Generation for Deep Packet Inspection of Binary Protocols. In *2017 15th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, 77–7709. https://doi.org/10.1109/PST.2017.00019
[5] MD Siam Hasan, Thomas Dean, Fahim T. Imam, Francisco Garcia, Sylvain P. Leblanc, and Mohammad Zulkernine. 2017. A Constraint-based Intrusion Detection System. In *Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems (ECBS '17)*. ACM, New York, NY, USA, Article 12, 10 pages. https://doi.org/10.1145/3123779.3123812
[6] John A. Hawkins. 1990. A Parsing Theory of Word Order Universals. *Linguistic Inquiry* 21, 2 (1990), 223–261. http://www.jstor.org/stable/4178670
[7] P. Hazel. 2006. *PCRE - Perl Compatible Regular Expressions.* http://www.pcre.org/
[8] Information Technology (ITU) 2015. *Abstract Syntax Notation One (ASN.1): Specification of basic notation.* Information Technology (ITU). http://www.itu.int/ITU-T/recommendations/rec.aspx?rec=12479
[9] David Law. 2016. IEEE Standard for Ethernet Amendment 7: Media Access Control Parameters, Physical Layers, and Management Parameters for 2.5 Gb/s and 5 Gb/s Operation, Types 2.5GBASE-T and 5GBASE-T. *IEEE Std 802.3bz-2016 (Amendment to IEEE Std 802.3-2015 as amended by IEEE Std 802.3bw-2015, IEEE Std 802.3by(TM)-2016, IEEE Std 802.3bq-2016, IEEE Std 802.3bp-2016, IEEE Std 802.3br-2016, and IEEE Std 802.3bn-2016)* (Oct 2016), 1–185. https://doi.org/10.1109/IEEESTD.2016.7600344
[10] David Law. 2019. IEEE Standard for Ethernet - Amendment 1:Physical Layer Specifications and Management Parameters for 2.5 Gb/s and 5 Gb/s Operation over Backplane. *IEEE Std 802.3cb-2018 (Amendment to IEEE Std 802.3-2018)* (Jan 2019), 1–206. https://doi.org/10.1109/IEEESTD.2019.8604150
[11] J. H. Lee, S. H. Cho, H. H. Lee, E. S. Jung, J. H. Yu, B. W. Kim, S. H. Lee, J. S. Koh, B. H. Sung, S. J. Kang, J. H. Kim, K. T. Jeong, and S. S. Lee. 2010. First Commercial Deployment of a Colorless Gigabit WDM/TDM Hybrid PON System Using Remote Protocol Terminator. *Journal of Lightwave Technology* 28, 4 (Feb 2010), 344–351. https://doi.org/10.1109/JLT.2009.2037979
[12] Sylvain Marquis, Thomas R. Dean, and Scott Knight. 2005. SCL: A Language for Security Testing of Network Applications. In *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '05)*. IBM Press, 155–164. http://dl.acm.org/citation.cfm?id=1105634.1105646
[13] Microsoft Corporation 2007. *[MS-SMB2]: Server Message Block (SMB) Protocol Versions 2 and 3* (53.0 ed.). Microsoft Corporation. https://msdn.microsoft.com/en-us/library/cc246482.aspx
[14] Object Management Group (OMG) 2010. *The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification (DDS-RTPS)* (2.1 ed.). Object Management Group (OMG). http://www.omg.org/spec/DDSI-RTPS/2.1/
[15] Terence Parr and Kathleen Fisher. 2011. LL(*): The Foundation of the ANTLR Parser Generator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 425–436. https://doi.org/10.1145/1993498.1993548
[16] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. *SIGPLAN Not.* 49, 10 (Oct. 2014), 579–598. https://doi.org/10.1145/2714064.2660202
[17] T. J. Parr and R. W. Quong. 1995. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810. https://doi.org/10.1002/spe.4380250705
[18] Terence J. Parr and Russell W. Quong. 1996. LL and LR Translators Need K>1 Lookahead. *SIGPLAN Not.* 31, 2 (February 1996), 27–34. https://doi.org/10.1145/226060.226066
[19] M. Patterson and D. Hirsch. 2014. Hammer parser generator. https://github.com/UpstandingHackers/hammer
[20] Mohamed Sami Rakha, Fahim T. Imam, and Thomas R. Dean. 2019. Generating a Real-Time Constraint Engine for Network Protocols. In *Information Security Theory and Practice*, Olivier Blazy and Chan Yeob Yeun (Eds.). Springer International Publishing, Cham, 44–60.
[21] Real-Time Innovations (RTI). 2010. Connext DDS Professional. https://www.rti.com/products/dds
[22] S. Sippu and E. Soisalon-Soininen. 2013. *Parsing Theory: Volume II LR(k) and LL(k) Parsing*. Springer Berlin Heidelberg. https://books.google.ca/books?id=xrqrCAAAQBAJ
[23] Valgrind 2019. *Memcheck: a memory error detector* (3.15.0 ed.). Valgrind. http://valgrind.org/docs/manual/mc-manual.html