Deep Grammar Optimization for Submessagae Structure of Network Protocol Parsers

A Technical Report

By Kyle Lavorato and Thomas Dean

COPASSTR20-2
Compass Group, Queen's University
Kingston, ON, Canada
June 2020

Deep Grammar Optimization for Submessagae Structure of Network Protocol Parsers

Kyle Lavorato
Queen's University
Kingston, Canada
13kpl2@queensu.ca

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

Abstract—**Network protocols may contain a structure where a single packet contains a header information and a set of multiple differing submessage elements. We propose an approach that will automatically detect this structure through the protocol's grammar and generate optimized parser code to decode it efficiently. This paper analyzes how the proposed optimization performs as part of an Intrusion Detection System, parsing traffic from the industry standard RTPS network protocol. We present a robust interface between the parser and packet analyzer provided at a deep level for each submessage to improve system performance.**

1. Introduction

The parsing of languages is a necessary activity in computer systems everywhere to create the software infrastructure of todays world. At some point in their creation, every program was parsed and compiled into its current state. With parsing being such a critical aspect of technology, there is a constant goal to discover more efficient ways to parse an input. Furthermore, of the subset of all grammars used in parsing, context-sensitive grammars have been used extensively. Parsing strategies such as a kernel with context-sensitive structured parse trees [1] or network protocol parsers dealing with deep packet inspection (DPI) [2] all use context-sensitive grammars. We introduce a grammar optimization algorithm for context-sensitive network protocol parsers. The algorithm will allow for efficient parsing of submessage structures in network protocols. The goal of our project is to create a grammar oriented optimization algorithm to increase the bandwidth of a network protocol parser. In this context the parser is designed to parse protocols that have been defined in SCL [3], a language designed by Marquis et al. as an extension to the ASN.1 [4] standard.

There have been considerable efforts in the research community to optimize parsers since the dawn of parsing theory [5], all with their own areas of optimization. Our optimization algorithm targets specifically network protocol parsers that parse the domain-specific language SCL that contains parsing constraints. These parsers are often used in limited networks as part of an Intrusion Detection System (IDS) as defined by Hasan et al. [6]. Since these parsers are parsing network traffic for intrusion detection they must be as efficient and as close to real time as possible, which is the motivation for this research.

1.1. Optimization Requirements

1.1.1. IDS Real Time Requirements. To create an effective IDS, it must be able to handle data in as close to real time as possible throughout its entire interface. The closer to real time it is able to process traffic, the earlier it is able to raise an alarm if an attack is detected on the network. When an alarm is raised early it is more likely that the intrusion can be dealt with before it deals significant damage.

1.1.2. Reliability. Any changes made to a parser must not affect its ability to produce a constant correct result from any input. This includes ensuring that our optimization does not conflict with any other elements of the parser. Conflicts can arise between multiple optimizations on the same structure. We must also ensure that no memory leaks are introduced into the environment as that will affect the parser's ability to run with zero down time.

1.1.3. Security. Alterations to the grammar of SCL can potentially create new possible vectors of attack in packets by allowing additional input. We must ensure that our optimization does not create any security vulnerabilities in the parser's ability to properly detect malformed and malicious packets.

1.1.4. Usability. The original parser by ElShakankiry et al. is designed to be simple to operate from the user's perspective. We must ensure that we follow this standard by not requiring any complex actions from the user to operate our new optimization.

2. Background

2.1. The TXL Source Transformation Language

TXL is a programming language designed for source to source transformation and rapid prototyping in software systems [7]. It allows the programmer explicit control over the interpretation, application, order and backtracking of parsing and rewriting of rules. TXL has evolved over time to become an industry standard general purpose source transformation tool. It is suited to a wide array of software reengineering tasks, including the Year 2000 problem of reprogramming billions of lines of commercial source code [8].

A TXL source transformation is a twofold process. First a context-free grammar set is defined for each source language being used in the transformation. 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. Rules and functions are written to match the contextual information of the input language using grammar elements and transform it to the specified target.

2.2. Semantic Constraint Language

SCL is a language that provides an extensive tool-set for defining binary network protocols and any limitations that are applied to them. SCL itself is an extension to ASN.1, an internet standard for defining binary protocols. Marquis et al. designed SCL to use the definition core of ASN.1 and extended it by adding XML style markup. The markup adds additional information and limitations to the protocols. There are three XML markup blocks added to the original ASN.1 grammar for `size`, `transfer` and `constraints` statements.

Considering the parser written by ElShakankiry et al. and our optimization, only the transfer block is under evaluation. The transfer markup specifies constraints that are relevant when decoding the data through parsing. There are three main transfer statements that the parser must consider when decoding any packet. The first two are `back` and `forward` blocks, which operate similarly. They both represent a conditional statement that must hold true during a parse, otherwise the packet is determined to be malformed and will fail. In a 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. The final transfer statement and the one of interest to our grammar optimization is the `callback` statement.

2.2.1. Callback Constraint. The callback statement is a parse termination command, which sends the decoded information from the parser using C function parameters. Its main purpose is not simply to terminate a parse, but to solve a large inefficiency within the SCL decoding process. In Protocol Tester by Marquis et al. [9], their decoding process is efficient when parsing protocols encoded with Basic Encoding Rules (BER) [10]. They experience difficulties when BER is not used, for example protocols such as OSPF [11], IGMP [12] and NFS [13]. This is due to the encoding of these protocols where they generally only specify a number of bytes that must be decoded for each field. This can result in graph structures like the partial decode example in Figure 1. Here we have a `GrmRec` with several `Field` elements, which inherit the base `Grm` class. This decoding flow creates a scenario where the graph must be walked post parse to discover the types of all the `Field` nodes. The walk must always be completed before the packet can be typed and sent to its post parse destination.

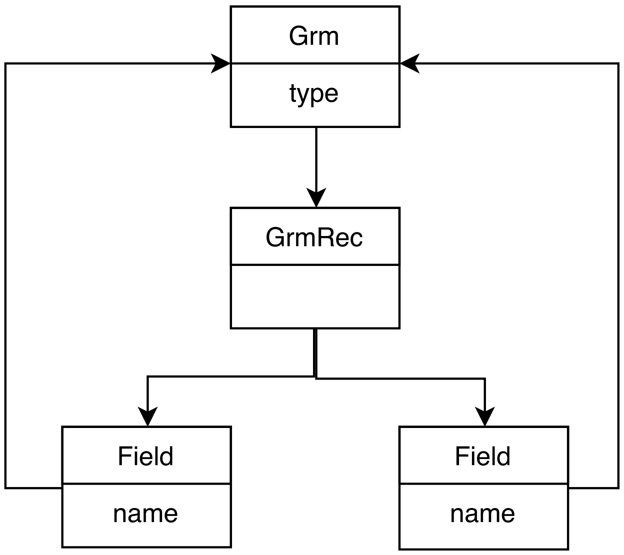

Figure 1: Flowchart of a partial SCL decode

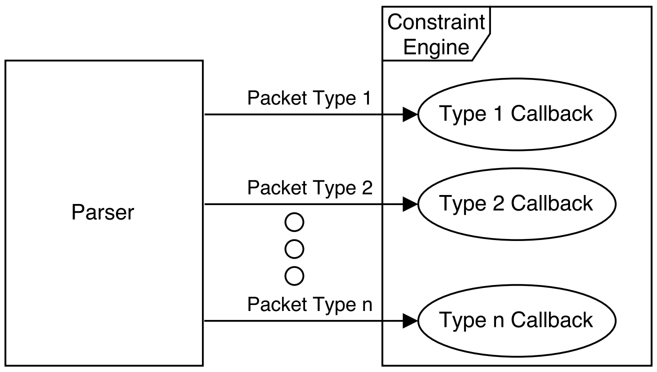

Figure 2: The original callback interface solution

The callback statement solves this post parse inefficiency by eliminating the need for the structure walk. When a callback constraint is indicated in a SCL transfer block, a typed callback function is generated at the end of the parse code for that definition. The typed callback function will pass the C structure for the specific protocol to the constraint engine. It indicates the type of the packet that the engine is receiving so it does not need to process its type. With unique naming, a parser for standard network traffic will create a scenario as described in Figure 2. Each packet type that must reach the constraint engine will have its own typed callback function to send decoded packets.

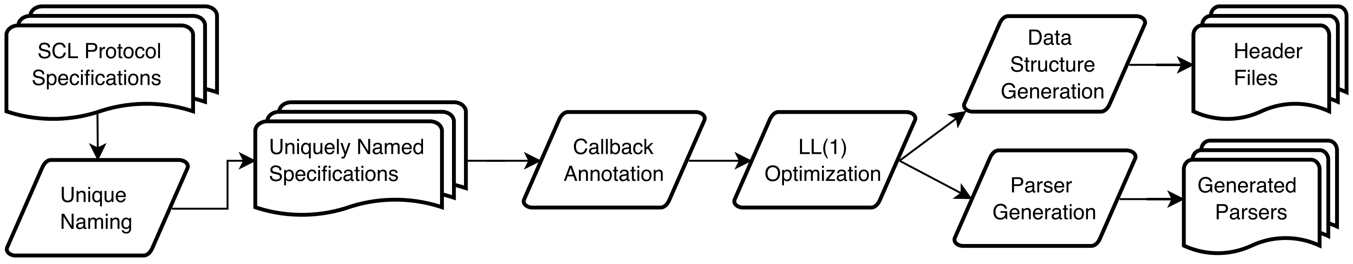

Figure 3: Parser generation flow chart with grammar optimization

2.3. Network Protocol Parser

TXL has been used as the backbone of the parser written by ElShakankiry et al. It is used in each phase of the annotation of the original SCL protocol source files, translating the source from SCL to SCL while adding markup to prepare it for code generation. The annotation first applies unique naming to all declarations and references by appending the name of the protocol to each. Next rule definitions are annotated with size and value markup and then evaluated for LL(1) optimization compatibility. Compatible rules are annotated with `@ optimizable` markup to indicate their status in the code generation phase.

Post markup phase, TXL is used to transform the SCL source to C code that parses each original source protocol. The annotated SCL source is first used in a SCL to .h transformation where a C header file is generated for the protocol. The header consists of structs to hold the data of each SCL rule definition in the protocol. Once the header is generated the full C source code is generated in the SCL to C transformation to create the parser for that protocol.

We have adapted this TXL backbone to place the markup for our optimization by adding a new step to the previous TXL pipeline. Once the post markup stage is reached the SCL to C transformation was altered to search for the new markup we added. The new markup transforms to new C source code for our optimized parse methods. Our new TXL generation pipeline is described in Figure 3.

3. Callback Efficiency

The callback transfer statement is an improvement over the original decoding method but it also suffers from a similar efficiency issue itself. Consider the RTPS protocol and its message structure [14]. A standard full RTPS packet has the structure shown in Figure 4. Each packet has a global header, which can be composed of multiple fields in a SCL description and a set of submessages. In a RTPS packet, the submessages are the component that hold the relevant data of the message. The header holds information identifying the packet as RTPS, its version number and the vendor that sent the message. This header information is relevant to each submessage and can impact the operation of the submessage contents. Each submessage field in a RTPS packet contains a set of any number of nine possible submessage types. Each

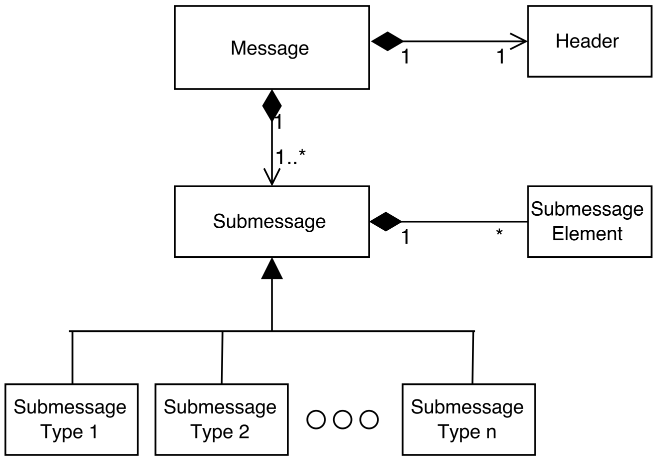

Figure 4: UML diagram depicting the submessage structure

type must be evaluated post decoding as separate messages as they have unique purposes and constraints.

The submessage structure of RTPS poses an almost identical problem to the inefficiency of Section 2.2.1. When the callback executes on a full RTPS packet, the receiving end has no information on the types of the submessages in the parsed packet. It must therefore walk the entire array of submessages and for each element, determine its type and then evaluate it combined with the header information. This poses a twofold avoidable resource drain on the receiving end of the constraint engine. The requirements of both walking the submessage structure each time any message needs to be typed and having to allocate memory for the storage of the array pose the issue. In order to solve this problem, we propose implementing a deeper level of callbacks that will be specific to each RTPS submessage type.

4. Grammar Extensions

4.1. Annotation Script

The proposed solution to the callback inefficiency requires an SCL annotation syntax to markup rules for the submessage callback code generation. We propose a simple three segment annotation to the existing callback transfer statement to markup rules with all necessary information for the code generation. Figure 5 describes the alteration

```
define transfer_statement
     [back_block]                          [NL]
   | [forward_block]                       [NL]
   | 'ALL 'BYTES 'USED                     [NL]
   | 'Callback [opt callback_annotation]   [NL]
end define

define callback_annotation
     [id] [id]
   | '@ [id]
   | '^ [id] [id]
   | 'Final
end define
```

Figure 5: Grammar changes for the optimization

to the SCL grammar with the addition of an optional `callback_annotation` to callback transfer statements. Each `callback_annotation` is designed to be as simple as possible and each one is used as markup on a specific segment of the submessage's definition process.

The annotation script is a three pass TXL transformation, with each pass adding markup to a different section of the submessage structure. The full SCL definition of the submessage structure that we are annotating is shown in Figure 6. For simplicity, the definition is a scaled down version of RTPS with only two submessage types. There must first be a rule definition where the final element is `SET OF` a user defined type. That user defined type must reference a type decision rule, indicating that the set is composed of multiple different types. The SCL grammar definition of `element_type` allow us to follow the reference in the first rule from the user defined type into the type decision rule. Then based on the grammar of the `type_decision` in the type decision rule we are able to again follow the references to each defined submessage definition.

The TXL script uses this grammar analysis to markup individual rules in each phase. An example of the SCL source of the scaled down RTPS definition post callback annotation markup is provided in Figure 7. This example indicates the context of each possible `callback_annotation`.

4.1.1. Pass One. The main grammar analysis occurs in the first pass of the TXL script. It searches for the pattern that matches the conditions for our optimization. The pattern we are trying to match is a rule definition where the final `element_type` is a `SET OF [id]` and the transfer block contains a `Callback`. If the pattern is found then we annotate the callback with our first annotation type "`@ [id]`," where the [id] is the user defined type's identifier in the `SET OF [id]`. This informs the code generator that we are initiating a deep callback on the specified type decision. Additionally, the [id] value from the annotation is saved for matching in the next pass.

4.1.2. Pass Two. The type decision rule is the focus of the second pass of the annotation process. The pattern to match is any type decision rule that has the name matching any saved identifier from the first pass. A callback statement will be added to the rule's transfer block with our second annotation type "$\wedge$ `[id] [id]`." This annotation begins with a '$\wedge$' symbol to keep the grammar unambiguous. The two [id] elements in the annotation are the name of the rule definition the original callback occurred in and the allocation name that represents the `SET OF` element from the same rule definition. This information is added to the annotation as the code generator requires the background information of the deep callback structure to generate the parser code. Each type identifier in the `type_decision` is then saved for matching in the next pass.

```
FULL ::= SEQUENCE {
    Header  HEADER (SIZE DEFINED),
    guidPrefix  GUIDPREFIX (SIZE DEFINED),
    subMsg  SET OF SUBMESSAGE (SIZE CONSTRAINED)
} (ENCODED BY CUSTOM)
<transfer>
    Back { Header.protoName == 'RTPS' }
    Forward { END(subMsg) }
    Callback
</transfer>

SUBMESSAGE ::= (ACKNACK | INFO_DST)

ACKNACK ::= SEQUENCE {
    kind       INTEGER (SIZE 1 BYTES),
    flags      INTEGER (SIZE 1 BYTES),
    nextHeader INTEGER (SIZE 2 BYTES),
    readerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
    writerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
    readerSN   SNSTATE  (SIZE DEFINED),
    counter    INTEGER (SIZE 4 BYTES)
}
<transfer>
Back {kind == 6}
Forward { ENDIANNESS == flags & 1 }
</transfer>

INFO_DST ::= SEQUENCE {
    kind       INTEGER (SIZE 1 BYTES),
    flags      INTEGER (SIZE 1 BYTES),
    nextHeader INTEGER (SIZE 2 BYTES),
    guidPfx    GUIDPREFIX (SIZE DEFINED) BIGENDIAN
}
<transfer>
Back {kind == 14} -- 0x0e
Forward { ENDIANNESS == flags & 1 }
</transfer>
```

Figure 6: Initial user written SCL code for a subset of RTPS

4.1.3. Pass Three. The rules defining the types in the phase two type decision are the focus of the third pass of the annotation process. Each rule that matches a saved [id] from phase two has a callback statement added to its transfer block. The callback is annotated with our third annotation type "`[id] [id]`." This annotation contains the same two [id] values as the annotation from the second pass. The annotation signals to the code generator that a callback function needs to be generated for that rule using the supplied [id] information and the definition of the rule itself.

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

SUBMESSAGE_RTPS ^ SUBMESSAGE ::= (ACKNACK_RTPS | INFO$DST_RTPS) < transfer >
    Callback ^ FULL_RTPS subMsg
</ transfer >

ACKNACK_RTPS ^ ACKNACK ::= SEQUENCE {
    kind_ACKNACK_RTPS ^ kind INTEGER (SIZE 1 BYTES),
    flags_ACKNACK_RTPS ^ flags INTEGER (SIZE 1 BYTES),
    nextHeader_ACKNACK_RTPS ^ nextHeader INTEGER (SIZE 2 BYTES),
    readerEnt_ACKNACK_RTPS ^ readerEnt ENTITYID_RTPS (SIZE DEFINED) BIGENDIAN,
    writerEnt_ACKNACK_RTPS ^ writerEnt ENTITYID_RTPS (SIZE DEFINED) BIGENDIAN,
    readerSN_ACKNACK_RTPS ^ readerSN SNSTATE_RTPS (SIZE DEFINED),
    counter_ACKNACK_RTPS ^ counter INTEGER (SIZE 4 BYTES)
} < transfer >
    Back {kind_ACKNACK_RTPS == 6}
    Forward {ENDIANNESS == flags_ACKNACK_RTPS& 1}
    Callback FULL_RTPS subMsg
</ transfer >

INFO$DST_RTPS ^ INFO$DST ::= SEQUENCE {
    kind_INFO$DST_RTPS ^ kind INTEGER (SIZE 1 BYTES),
    flags_INFO$DST_RTPS ^ flags INTEGER (SIZE 1 BYTES),
    nextHeader_INFO$DST_RTPS ^ nextHeader INTEGER (SIZE 2 BYTES),
    guidPfx_INFO$DST_RTPS ^ guidPfx GUIDPREFIX_RTPS (SIZE DEFINED) BIGENDIAN
} < transfer >
    Back {kind_INFO$DST_RTPS == 14}
    Forward {ENDIANNESS == flags_INFO$DST_RTPS& 1}
    Callback FULL_RTPS subMsg
</ transfer >
```

Figure 7: SCL code for a subset of RTPS that has been annotated by the grammar optimization

4.1.4. User Control. There may be scenarios where the user desires to not use our optimization as they may possibly need the packets sent from the parser in the standard callback form. In anticipation of this we have added one additional `callback_annotation` type with the "`Final`" annotation. If the TXL script matches a `Callback Final` statement in a transfer block then it does not attempt to optimize it. Instead it removes the annotation to preserve the callback's compatibility with the remainder of the TXL pipeline. The intent of this extra annotation is to enable our optimization to be as user friendly as possible. The user only must learn one additional SCL line of the `Callback Final` when using our optimization. They are also given full control over the location in which they wish to use the optimization. It will only activate where there is a case to optimize, removing any requirements of grammar recognition from the user.

4.2. LL(1) Optimization Compatibility

If an SCL definition for type decisions in a protocol matches the optimization conditions described by ElShakankiry et al. then a LL(1) look-ahead parser is generated instead of the normal backtracking parser. The LL(1) optimization condition can interact with the same type decision rules that we optimize by grammar, causing LL(1) to be incompatible with our grammar optimization. We have only optimized the backtracking parser, therefore there is no case in the code generation that will match both optimization's markup in the same rule.

The RTPS protocol has no possible scenario where both optimizations would be required under standard use. Therefore, for practically reasons we have chosen to not optimize the look-ahead parser. We remove our optimization annotation instead if we match that the SCL has been marked up with both annotation types. The check is necessary due to the possibility of special cases where it may be desired to have a callback in an unusual location. This removal is performed in the same TXL script that places the LL(1) markup, which must be performed before any C code is generated, as shown in Figure 3. Additionally, the markup removal is written for future extensions with it being simple to expand to have a simple toggle if the parser is to be used with protocols that require both.

```
typedef struct {
    HEADER_RTPS header;
    GUIDPREFIX_RTPS guidprefix;
    unsigned long submsglength;
    unsigned long submsgcount;
} FULL_RTPS;
```

Figure 8: New parent rule struct definition

5. Parser Generation

Once the SCL source has been annotated, it is prepared for the code generation transformation. First the header file is created, following the standard generation described by ElShakankiry et al. When the "`@ [id]`" annotation is matched during header generation a modified parent struct is generated as shown in Figure 8. The element in the [id] field will be omitted from the struct for that rule but the *length* and *count* fields are still included. The pointer array of submessages no longer requires storage as the deep callbacks will pass the specific typed struct instead of a pointer to a union struct. When the "`[id] [id]`" annotation is matched during header generation, a function header definition will be generated for the deep callback. The function headers are named based on the name of the individual rule and the protocol's name. Their parameters will always be a set of three pointers; A struct of the parent type from the annotation's first [id] field, a struct of the defining rule's type and the PDU type. In the case of the ACKNACK submessage, the full function header would be `ACKNACK_RTPS_callback(FULL_RTPS *full_rtps, ACKNACK_RTPS *acknack_rtps, PDU *thePDU);`.

The C source code generation follows the standard generation described by ElShakankiry et al. Additional pattern matching has been added for rules with annotated callbacks as none of the existing rules will match and generate code. We have developed two different methods of parsing the submessage structure which differ in both format and timing of execution of the deep callback functions. We have named these the *Walker Parse* and *Immediate Parse*. Each method has complimenting strengths, weaknesses and performance benchmarks, causing them both to be relevant solutions in differing scenarios.

5.1. Callback Interface

The new deep callback interface to the constraint engine is designed to correct the efficiency issues identified in Section 3. The callback interface is currently implemented as custom written C code, as the generation of constraints is not yet automated. The callback functions are designed to activate and run the constraint engine validation on the passed in packet structures. The original inefficient interface function for our scaled down RTPS protocol definition is shown in Figure 9a. Our new deep callback interface shown in Figure 9b, runs the same constraint engine code as the original. Since each function already knows the type of the passed in packet, unlike in the initial version, the constraint code can be run instantly instead of requiring to type the message first.

5.2. Walker Parse

The walker parse method is designed for systems where more strain can be placed on the parser while providing a slight reduction in work to the receiving end of the decoded packets. The theory behind this method is to take the work that was originally completed in the constraint engine side and move it to the parser since the constraint engine is the current bottleneck. Previously when a parsed packet was sent to the constraint engine, the submessage array was walked to type the submessage and then correctly process it.

Our source code structure does not change the way the submessage element is parsed. We create a local pointer variable to hold the submessage array since it is no longer a struct member. It is then allocated and filled by the recursive parse for `SET OF` elements designed by ElShakankiry et al. This change is triggered by the `@ [id]` callback annotation in the parent rule definition. The proper post array parse constraint checks are still performed on the local variable to ensure no security issues may arise malformed packets. The walker function for this local array variable, shown in Figure 10 is then called after the local array has been tested. The walker function itself is generated piece by piece with each rule containing the `[id] [id]` callback annotation generating a segment. Each generated segment is added to a collection of statements that match the parent identifier value. The walker function is a switch statement inside a loop that has a case for each possible submessage type. Each iteration the type value is retrieved from the current submessage under evaluation and then used as the switch statement parameter. It is compared to predefined constant values for each submessage. Each case consists of the same three statements; The deep callback to send the packet pieces, a deep free function for that submessage to prevent memory leaks and a break statement as there can only ever be one possible choice in the switch statement. From a security standpoint, the switch statement has a default case which fails the parse in the event of the submessage's type not matching any of the defined values, indicating a malformed packet.

To prevent memory leaks the method of freeing allocated memory was required to be changed. Previously the allocated submessage array would be freed through the member in the main packet struct in another walking structure. Since that has been removed from the struct and replaced with a local variable, that local variable must be freed before the parse of the current packet ends. Free functions are now created for each rule that has the `[id] [id]` callback annotation as they represent a message type that will always have allocated memory. In these new free functions, the object requiring freeing is passed in as a pointer instead of by reference for efficiency. Therefore all references are changed to use the member by pointer arrow operator, '`->`'

```
void FULL_RTPS_callback(FULL_RTPS *r, PDU *thePDU)
{
  struct HeaderInfo *h = thePDU->header;
  for(int i = 0; i < r->submsgcount; i++)
  {
    if(r->submsg[i].type == ACKNACK_RTPS_VAL)
    {
        ...
    }
    else if(r->submsg[i].type == INFO$DST_RTPS_VAL)
    {
        ...
    }
  }
}
```

(a)

```
void ACKNACK_RTPS_callback (FULL_RTPS *full_rtps,
    ACKNACK_RTPS *acknack_rtps, PDU *thePDU) {
        ...
}

void INFO$DST_RTPS_callback (FULL_RTPS *full_rtps,
    INFO$DST_RTPS *info$dst_rtps, PDU *thePDU) {
        ...
}
```

(b)

Figure 9: (a) Original callback interface function. (b) Optimized deep callback interface functions

```
bool walk_SUBMESSAGE_RTPS (FULL_RTPS *full_rtps, SUBMESSAGE_RTPS *submsg, PDU *thePDU) {
    uint32_t type;
    for (int i = 0; i < full_rtps->submsgcount; i++) {
        type = submsg[i].type;
        switch (type) {
        case INFO$DST_RTPS_VAL :
            INFO$DST_RTPS_callback (full_rtps, &submsg[i].ptr.info$dst_rtps, thePDU);
            freeINFO$DST_RTPS (&submsg[i].ptr.info$dst_rtps);
            break;
        case ACKNACK_RTPS_VAL :
            ACKNACK_RTPS_callback (full_rtps, &submsg[i].ptr.acknack_rtps, thePDU);
            freeACKNACK_RTPS (&submsg[i].ptr.acknack_rtps);
            break;
        default :
            return false;
        }
    }
}
```

Figure 10: Parse structure for the walk optimization method

instead of the member by reference dot operator, '.'. The remainder of the existing free method for a packet of the protocol still functions as before but does not include freeing any of the types handled by the deep frees.

Walking the submessage array in the parser removes the maximum amount of strain from the constraint engine that would be caused by the submessages. Due to the parse of each submessage being verified as successful before the walk is started, the constraint engine can begin processing each submessage as soon as it is received. The constraint engine will never need to backtrack due to a parse failing after only a portion of the submessages have been sent. The benefit to the constraint engine comes at a cost to the parser efficiency. Memory allocation for the entire array of submessages requires both time and resources, reducing the bandwidth of the parser. The requirement of walking the entire array at callback time also reduces the bandwidth. Although since callback and free are grouped together the extra walk that was required at free time is now avoided.

5.3. Immediate Parse

The immediate parse method is designed for systems where it is desired to have the parser and constraint engine running as efficiently as possible combined. The theory behind this method is to send data to the constraint engine as soon as a submessage is successfully parsed. By sending submessages as soon as they are available, the constraint engine is able to begin processing them immediately.

As with the previous method we do not change the way the submessage element is parsed. We begin by creating a local pointer to the submessage union struct when the `@ [id]` callback annotation is matched. We then create a modified version of the recursive parse to interact with the local pointer. Since we intend to send the submessages immediately after they are parsed, there is no need to store them in an array. By eliminating the memory allocation and assignment of the array there is large potential savings when running tens of thousands of packets through the parser. The recursive parse retains its structure as shown in Figure 11, using the same termination conditions as before to parse all submessages. The non-saving recursive parse is generated

```
SUBMESSAGE_RTPS *parseSetOfSUBMESSAGE (FULL_RTPS *full_rtps, PDU *thePDU, int n, int *size, char *progname,
                                        uint8_t endianness) {
    SUBMESSAGE_RTPS submsg;
    if (!parseSUBMESSAGE (full_rtps, &submsg, thePDU, progname, endianness)) {
        if (n == 0) {
            *size = 0;
            return NULL;
        }
        *size = n;
        return NULL;
    }
    else {
        SUBMESSAGE_RTPS *result = parseSetOfSUBMESSAGE (full_rtps, thePDU, n +1, size, progname, endianness);
        result = &submsg;
        return result;
    }
}
```

Figure 11: Non-saving recursive parse function for SET OF elements

```
bool parseSUBMESSAGE (FULL_RTPS *full_rtps, SUBMESSAGE_RTPS *submessage_rtps, PDU *thePDU, char
                        *progname, uint8_t endianness) {
    unsigned long pos = thePDU->curPos;
    unsigned long remaining = thePDU->remaining;
    if (parseACKNACK (&submessage_rtps->ptr.acknack_rtps, thePDU, progname, endianness)) {
        full_rtps->submsglength = thePDU->curPos - full_rtps->submsglength;
        full_rtps->submsgcount = 1;
        ACKNACK_RTPS_callback (full_rtps, &submessage_rtps->ptr.acknack_rtps, thePDU);
        freeACKNACK_RTPS (&submessage_rtps->ptr.acknack_rtps);
        return true;
    }
    thePDU->curPos = pos;
    thePDU->remaining = remaining;
    if (parseINFO$DST (&submessage_rtps->ptr.info$dst_rtps, thePDU, progname, endianness)) {
        full_rtps->submsglength = thePDU->curPos - full_rtps->submsglength;
        full_rtps->submsgcount = 1;
        INFO$DST_RTPS_callback (full_rtps, &submessage_rtps->ptr.info$dst_rtps, thePDU);
        freeINFO$DST_RTPS (&submessage_rtps->ptr.info$dst_rtps);
        return true;
    }
    return false;
}
```

Figure 12: Parse structure for the immediate optimization method

from the $\wedge$ `[id] [id]` callback annotation on the type decision rule. Upon termination when all the submessages are parsed, the final element is returned in the pointer, allowing for testing of the post parse constraints to ensure the parse was successful. The function `parseSUBMESSAGE` which is called from the recursive parse is where the deep callbacks occur. It is generated piece by piece from the rules with `[id] [id]` callback annotation as a backtracking parser. Our optimization changes the code that executes when the parse of a submessage is successful, as shown in Figure 12. The length and count fields of the parent struct are set, the deep callback function is called and the submessage that was parsed is freed using a deep free function.

Avoiding unnecessary malloc operations and walking of array structures allows the parser to execute as efficiently as possible. Not only is this benefit present, but sending submessages to the constraint engine as soon as they are parsed allows for earlier processing. Additionally, by sending submessages immediately as they are available there is a natural extension to a parallel processing producer/consumer queue system. Despite the benefits of this system there is a potential weakness in the method. A parse can fail due to a malformed or dangerous submessage once one or more have already reached callback and been sent to the constraint engine. In this scenario, the engine must backtrack and remove those submessages from a queue or processing as they are potentially dangerous. The parser supplies the pass and fail results for the packet but there is no current method to apply this result to backtracking as we have focused on the parser grammar optimization.

6. Validation

We evaluate the generated optimized parse methods by comparing to the performance of the base Network Protocol Parser by ElShakankiry et al. We follow the existing IDS

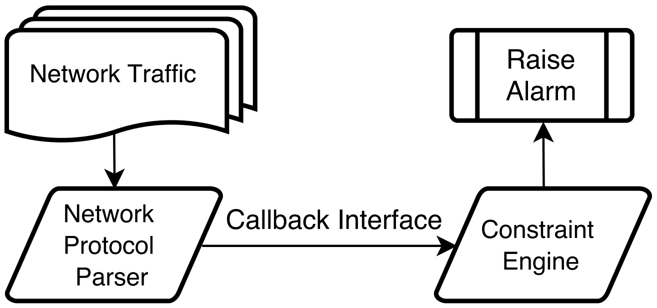

Figure 13: The overall IDS interface

interface as described in Figure 13. The parser decodes and verifies network traffic in the form of packets, sending them to the constraint engine through callbacks. The constraint engine then analyzes them against constraints of what standard network traffic should look like. If they fail constraint checking then the engine would raise an alarm to the rest of the system.

Testing was completed by measuring the performance of both the parser in isolation and the full IDS interface, using a 1668 MB packet capture of realistic network data. The packet capture is generated using the RTI version of the Data Distribution Service (DDS) [15]. It generates RTPS data and also includes IGMP, ARP and NTP packets to simulate a realistic air traffic control limited network. A full 100 parses were run on each of the three IDS builds in both parser isolation and full system interface using the `pcapparse` executable. An average value was calculated from the 100 trials since parser performance can vary depending on system load. It is especially difficult to keep constant system load through all trials over an extended period. The resulting standard deviation of the bandwidth measurement is an average of *175.09 (Mbit/s)* between all the builds. All tests were performed on a quad-core CPU running at 2.60 GHz.

Tables 1 and 2 show the performance differences between each parse method when run in parser isolation and complete IDS interface respectively.

TABLE 1: Performance of Parser

Algorithm	Run Time (s)	Bandwidth (Mbit/s)
Standard Parse	9.6	1402.17
Walker Parse	9.4	1432.37
Immediate Parse	9.0	1509.62

TABLE 2: Performance of Parser and Constraint Engine

Algorithm	Run Time (s)	Bandwidth (Mbit/s)
Standard Parse	9.8	1379.97
Walker Parse	9.6	1408.91
Immediate Parse	9.3	1457.26

The results validate the strengths and weakness of each parse method as outlined in Sections 5.2 and 5.3. Each method provides a clear increase to the performance of the entire system, improving the net bandwidth. The walker parse in Table 3 is an overall *2.13%* increase in bandwidth, while the immediate parse in Table 4 is an overall *6.63%* increase in bandwidth. The bandwidth increase percentages are all calculated from the base Network Protocol Parser as a reference.

TABLE 3: Performance Increase of the Walker Parse

Interface	Speedup (%)
Parser Only	2.15
Full System	2.10

TABLE 4: Performance Increase of the Immediate Parse

Interface	Speedup (%)
Parser Only	7.66
Full System	5.60

Considering the performance and strengths of the parse methods, the immediate parse is the better of the optimizations. Although both have applicable scenarios where they should be used we suggest the immediate parse to be generally selected when our grammar optimization is applied. Although our suggested optimization applies a 6.63% increase to performance, it is still possible to optimize further, especially since the generated parser only has two current optimizations. There is an extensive drive to create optimizations in the research community and we discuss similar grammar optimizations in the following section.

7. Related Work

As the problem of parsing theory and efficient parsing is a classic one, there are countless solutions of vastly varying parser optimizations. We review solutions that share similarities with our grammar optimization such as ANTLR [16], ANTLR4 [17], YAKKER [18] and Zebu [19]

7.1. ANTLR LL(*) Parser

ANTLR is a parser generator that introduces a LL(*) parsing strategy, and an associated grammar analysis system to construct the LL(*) parsing decisions. LL(*) is a parsing optimization that throttles up from the conventional $k \geq 1$ to an arbitrary lookahead. It will then fail over to a backtracking parse depending on the complexity of the parsing decision. 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 grammar.

The parser itself implements a left-to-right one pass parse, using lookahead DFAs. The DFAs match the nonterminal input to predict which production the parser should expand. The generation analyzes each nonterminal in the grammar with multiple productions to construct the DFAs. They propagate through the grammar, applying an algorithm that searches for a LL-regular partition block for the nonterminal. If the grammar is not LL-regular it has backup algorithms that will derive a DFA in a less efficient manner. ANTLR4 is a new version that improves the work of the previous LL(*) parse, introducing the ALL(*) parsing strategy. In the ALL(*) strategy they innovate 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.

Unlike in our optimization that analyzes the grammar for a particular structure, ANTLR translates the entire grammar into a decision structure. We do not attempt such a large-scale transformation and rather improve a smaller inefficiency. They successfully implement an improved version of the LL(1) optimization that was designed by ElShakankiry et al. Since our grammar analysis algorithms target separate areas than LL(*) and ALL(*), the optimizations can be adapted to be compatible when implemented in a single system

7.2. YAKKER Attribute-Directed Parsing

YAKKER is a parsing engine that uses a data-dependent grammar parsing strategy to satisfy the needs of modern programmers and data processing applications. It is a full parsing engine, but in this section, we are focusing on an optimization that YAKKER applies to their parsing algorithm, named attribute-directed parsing. Similar to our work, YAKKER uses grammar elements to form the basis of an enhancement to their parsing algorithm. Their data-dependent grammar contains nonterminals that can capture input substrings to direct the parser's behaviour. This occurs commonly in the binary data of the network protocols that we parse, where they contain messages consisting of the length of the data field followed by the data. To optimize their parsing process, they convert their grammar definitions for these structures into a set of bindings of the length constraint. Therefore, the constraint values on the data length are already set and can be evaluated during the parse for efficiency. We employ a similar strategy when we annotate specific identifiers into our callback annotations so they can be evaluated during transformation time. Like with YAKKER if the information was not prepared beforehand there would be an inefficiency in determining the information during a semantic analysis phase.

7.3. 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. The added annotations allow for domain-specific optimizations to reduce memory usage of a Zebu-based application. For example, a `lazy` annotation gives the application control of when the field is to be parsed. Zebu uses a standard input format of ABNF with only slight modifications, whereas we use a heavily annotated version of the industry standard ASN.1. This allows Zebu to be much easier for a first-time developer to pick up and begin using comparatively. Instead of a pattern matching tool like TXL in our optimization, Zebu annotations are processed by the PCRE library [20] as an input of regular expressions and transformed into stub functions. The use of a library provides a much lower learning curve for other researches to extend their work in the future. Since our work uses the rather tricky and powerful language of TXL, it has a much higher learning curve to begin future additions. Zebu employs a parsing optimization that applies a grammar algorithm much like ours. If elements are annotated with a `nocheck` then the developer is notifying the application that the element requires no validation. When their generation engine matches the annotation it then it identifies the tokens before and after the annotated element through a grammar analysis. They are used as the bounds for the optimization. All the grammar elements in between the bounds are merged into one decode state as they need not be validated after the decode. This therefore reduces the amount of validation code that need be generated and executed for each packet. Although this optimization is not applicable in most IDS instances due to the need to verify traffic, it could still be an interesting addition for some specific scenarios that could be encountered by our system.

8. Future Work

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

8.1. LL(1) Compatibility

There potentially exists type decisions in other protocols where optimization by our grammar optimization and the LL(1) look-ahead parse are both possible. In that case it would be desired to have both optimizations active to generate the most efficient parser. If such a scenario is presented it would be necessary to create another pattern match in the TXL C code generator for both annotation markups in an SCL definition. A C code sequence that combines the LL(1) and desired grammar parse method for Walker or Immediate would also need to be derived.

8.2. Deeper Grammar Analysis

In it's current form, the grammar analysis only searches for the submessage structure on a single rule basis. In RTPS there is the possibility of an SCL definition using an intermediary user defined type containing the `SET OF SUBMESSAGE` element instead of adding it directly in the

`RTPS_FULL` rule. In this case our optimization would not detect the structure as it does not perform a deep type analysis. An extension to the annotation script would be to alter the first pass to use a reverse FOLLOW set analysis for deep type checking. By completing this analysis, it would allow the user to define a protocol in SCL any way they choose instead of requiring the structure to be in one rule to use our optimization.

8.3. Parallel Processing of Deep Callback

The deep callback interface in its current state provides optimal conditions for a parallel processing producer/consumer queue to be implemented in the constraint engine side of the IDS interface. Submessages are now being sent as individual packets, which no longer require special type processing in the callback functions. They can therefore be placed in a producer queue directly through the callback functions for the constraint engine to consume. The immediate parse method supports this idea further by populating the producer queue with entries faster than the walker to reduce the duration the producer queue may be empty.

By running the parser as a producer and constraint engine as a consumer on separate threads it will no longer pause the execution of the parser while the constraint engine analyses the decoded packet. The parser will be able retain the faster bandwidth of Table 1 when producing as it will run in isolation. The constraint engine will also likely generate a bandwidth increase through its ability to process a continuous queue instead of processing a packet and then sleeping until the next one is provided.

Acknowledgment

This research is funded in part by the National Science and Engineering Research Council of Canada (NSERC), the Department of National Defence (DND), Canada, and the Ontario Research Fund (ORF), Canada

References

[1] G. Zhou, M. Zhang, D. Ji and Q. Zhu, "Tree kernel-based relation extraction with context-sensitive structured parse tree information," in *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL), June 28-30, 2007, Prague, Czech Republic*, 2007 pp. 728-736. [Online]. Available: http://www.anthology.aclweb.org/D/D07/D07-1.pdf#page=762

[2] A. ElShakankiry, T. Dean, "Context sensitive and secure parser generation for deep packet inspection of binary protocols," Queen's University, Kingston, 2017.

[3] 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

[4] Information Technology (ITU), "Abstract Syntax Notation One (ASN.1): Specification of basic notation." [Online]. Available: http://www.itu.int/ITU-T/recommendations/rec.aspx?rec=12479

[5] J. A. Hawkins, "A Parsing Theory of World Order Universals," *Linguistic Inquiry*, vol. 21, no. 2, pp. 223-261, Spring, 1990. [Online]. Available: http://www.jstor.org/stable/4178670

[6] 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

[7] J. R. Cordy, The TXL source transformation language, *Science of Computer Programming*, vol. 61, no. 3, pp. 190-210, 2006. [Online]. Available: https://doi.org/10.1016/j.scico.2006.04.002

[8] 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*, vol. 44, no. 13, pp. 827-837, 2002. [Online]. Available: https://doi.org/10.1016/S0950-5849(02)00104-0

[9] S. Marquis, T. R. Dean and S. Knight, "Packet decoding using context sensitive parsing," in *CASCON '06 Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research, October 16-19, 2006, Toronto, Ontario, Canada*, 2006, no. 20. [Online]. Available: http://doi.acm.org/10.1145/1188966.1188993

[10] Information Technology (ITU), "ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)," [Online]. Available: http://www.itu.int/ITU-T/recommendations/rec.aspx?rec=12483

[11] J. Moy, "OSPF Version 2," July 1991. [Online]. Available: https://tools.ietf.org/html/rfc1247

[12] B. Cain et al., "Internet Group Management Protocol, Version 3," October 2002. [Online]. Available: https://tools.ietf.org/html/rfc3376

[13] B. Callaghan, B. Pawlowski, P. Staubach, "NFS Version 3 Protocol Specification," June 1995. [Online]. Available: https://tools.ietf.org/html/rfc1813

[14] Object Management Group (OMG), *The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification (DDS-RTPS)*, 2.1 ed. Real-Time Innovations, Inc., 2010. [Online]. Available: http://www.omg.org/spec/DDSI-RTPS/2.1/

[15] Real-Time Innovations (RTI), "Connext DDS Professional." [Online]. Available: https://www.rti.com/products/dds

[16] T. Parr, 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

[17] T. Parr, S. Harwell, K. Fisher, Adaptive LL(*) parsing: the power of dynamic analysis, in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, Portland, Oregon, USA, October 20-24, 2014*, 2014, pp. 579-598. [Online]. http://doi.acm.org/10.1145/2714064.2660202

[18] T. Jim, Y. Mandelbaum, D. Walker, Semantics and algorithms for data-dependent grammars, in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, 2010, pp. 417-430. [Online]. Available: http://doi.acm.org/10.1145/1707801.1706347

[19] L. Burgy, L. Reveillere, J. Lawall and G. Muller, Zebu: a language-based approach for network message processing, in *IEEE Transactions on Software Engineering, July-August 2011*, 2011, vol. 37, no. 4, pp. 575-591, 2006. [Online]. Available: https://doi.org/10.1109/TSE.2010.64

[20] P. Hazel, "PCRE - Perl Compatible Regular Expressions," 2006. [Online]. Available: http://www.pcre.org/