Targeted Parsing for Text-based Protocols

A Technical Report

By Mohammad Salloum, Fahim Imam, and Thomas Dean

COPASSTR20-4
Compass Group, Queen's University
Kingston, ON, Canada
June 2020

Targeted Parsing for Text-based Protocols

Mohammad Salloum
Queen's University
Kingston, Ontario, Canada
m.salloum@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

In this paper, we present a parser generation framework for text-based network protocols. A general ANTLR grammar is used to parse the messages and isolate the key part of each message that distinguishes the message type. Once the category is identified using a regular expression, the application-specific part of the message is parsed using an application specific subgrammar, we call this a targeted parsing approach. We present a specification language to automatically generate our parsers. We create a Constraint Engine prototype to demonstrate how we can use the parsers we generated in an intrusion detection scenario.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

network protocols, protocol parser, grammar

ACM Reference Format:
Mohammad Salloum, Fahim T. Imam, and Thomas R. Dean. 2019. Targeted Parsing for Text-based Protocols. In *AsiaCCS '20: 15th ACM ASIA Conference on Computer and Communications Security, June 01–05, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Intrusion detection systems (IDSs) play a crucial role in network security. To effectively detect intrusions, an IDS has to read and analyze network traffic. Therefore, an important module of an IDS is the parser that will parse the data found in network traffic. ElShakankiry et al. [1] worked on a parser to be used in an IDS. The parser that ElShakankiry et al. developed works on binary network protocols only. When it comes to IDSs, it is important to be able to parse network traffic running over text-based protocols. We develop a framework that allows us to parse traffic running over any text-based network protocol. Then we pair our parsers with a Constraint Engine to demonstrate how the parsers may be applied in intrusion detection.

Our parsing approach is similar to ElShakankiry et al. [1]. They use a formal language to model binary protocols such as RTPS [2]. They generate a parser from the protocol model. Our approach leverages the ANTLR parser [3] to parse text based protocols such as HTTP [4] and SMTP [5].

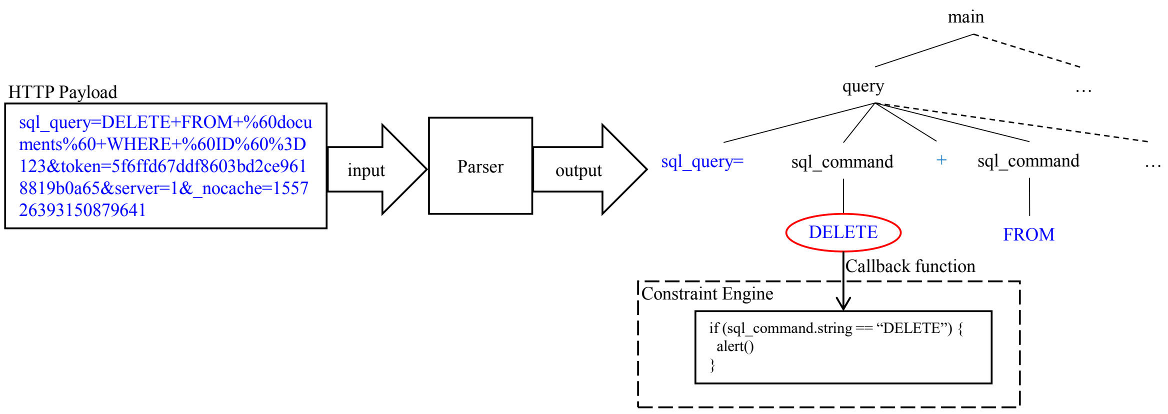
Our contribution is a framework that categorizes messages and then applies an application-specific parser to each message. These application-specific parsers allow the extraction of elements of interest in a protocol message. For example, when analyzing an application that utilizes HTTP such as PhpBB [6], our approach uses a general grammar to parse HTTP messages, and then a set of PhpBB specific grammars to parse each of the distinct application messages. We also provide a prototype of a Constraint Engine and use it to show how the output of our generated parsers can be used for intrusion detection.

We organize the paper as follows. Section 2 provides background knowledge needed to follow the rest of this paper. Section 3 discusses the overall workflow and parser architecture. In Section 4, we evaluate our parser as a standalone application and as a part of a system that contains a Constraint Engine. We present related work in Section 5. We conclude this paper and discuss the scope of our future work in Section 6.

2 BACKGROUND

Network engineers must detect network intrusions before any damage happens. Hasan et al. [7] developed an IDS targeted at limited networks. Such networks have a limited number of protocols that are running between their hosts. Hasan et al. took advantage of this property to detect intrusions. They were able to define the normal behavior of a network and detect anomalies by evaluating constraints. A constraint-based IDS [7] is feasible in limited computer networks since the network engineers are familiar with all the protocols. Examples of limited computer networks are low-risk, high-profile networks such as Air Traffic Control and Industrial Control networks. This approach is less feasible for computer networks connected to the internet where the network protocols are unpredictable. In these dynamic networks, a signature-based IDS [8] is more feasible. These systems detect intrusions by looking for signatures of attacks that have been previously discovered and publicized. A constraint-based IDS in such scenarios will generate false positives, defeating the purpose of an IDS.

The IDS developed by Hasan et al. in its current state only supports binary protocols such as the RTPS [2] protocol. The IDS needs to support text-based protocols to detect intrusions at the application level. An example application would be a network monitoring tool that runs on a browser and communicates with its server via HTTP. Applications built on top of text-based protocols represent a threat vector which intruders may manipulate.


HTTP Payload
sql_query=DELETE+FROM+%60documents%60+WHERE+%60ID%60%3D123&token=5f6ffd67ddf8603bd2ce9618819b0a65&server=1&_nocache=155726393150879641
input
Parser
output
main
query
...
sql_query=
sql_command
+
sql_command
...
DELETE
FROM
Callback function
Constraint Engine

```
if (sql_command.string == "DELETE") {
  alert()
}
```

Figure 1: Parsing and Constraint Check Example.

2.1 Example Attacks

Stuxnet [9] was able to breach into the air-gapped network of a nuclear facility in Iran. It was able to hop between different computer networks and USB drives until it infected the USB drive of one of the workers of the targeted nuclear facility. Once Stuxnet was inside the facility's computer network, it attacked programmable logic controllers (PLCs) that controlled the uranium-enriching centrifuges. It then proceeded to overload those centrifuges. At the same time, Stuxnet deceived the employees of the facility by modifying their monitoring system to show that there were no problems with the centrifuges.

What made Stuxnet so effective is that it used four zero-day vulnerabilities. A zero-day vulnerability is a vulnerability in a software that has not been discovered and published and therefore, there is no patch or fix for it. Stuxnet made use of zero-day vulnerabilities to traverse networks and remain undetected long enough to reach its target and execute its malicious payload. After Stuxnet, more worms targeting limited networks appeared such as Duqu, Gauss and Flame [10]. Zero-day attacks can not be detected with a signature-based IDS but may be detected with a constraint-based IDS.

2.2 Use Case Scenario

Consider the following use case with phpMyAdmin [11]. phpMyAdmin is a database management platform allowing users to interact with multiple databases and execute queries on them. A malicious user may tamper with a database through phpMyAdmin by using SQL [12] ALTER or DELETE statements. Listing 1 is an example of a typical HTTP message from a client to the phpMyAdmin server embedding SQL query statements.

```
 1 POST /phpmyadmin/lint.php HTTP/1.1
 2 Host: 127.0.0.1
 3 Accept-Language: en-US,en;q=0.5
 4 Accept-Encoding: gzip, deflate
 5 X-Requested-With: XMLHttpRequest
 6 Content-Length: 135
 7 Connection: keep-alive
 8 More HTTP headers...
 9 sql_query=DELETE+FROM+%60documents
10 %60+WHERE+%60ID%60%3D123&token=5f6
11 ffd67ddf8603bd2ce9618819b0a65&serv
12 er=1&_nocache=1557263931508796416
```

Listing 1: phpMyAdmin Query Request.

The SQL statements in the payload of the HTTP element is not in an intuitive format. Another problem is that not all HTTP messages going through the network are phpMyAdmin traffic. Some may even contain different application data that we need to parse with a different custom parser. To handle those cases, we developed an approach that supports message categorization and targeted parsing.

To categorize a message, we first parse the message with a generic protocol parser. We use the generic parser to identify one or more elements within the message that distinguishes that message from other messages on the network. The elements are matched to regular expressions to determine the category of the message. The message element containing the application data is then extracted and labeled by its category. We parse the extracted data with a parser specific to the message's category. In the example, we match the URL on line 1 of Listing 1 to the regular expression: `".*lint\\.php"`. If the regular expression matches the URL, we proceed to parse the message body (lines 9 to 12) with a parser specific to that message body format. The specific parser can extract the query values we are interested in.

After we parse the application data, we can check for constraints on that data. A constraint is any restriction or limitation we specify on application data. In our example, one of the constraints could be that none of the queries should be an ALTER or DELETE statement. Therefore, whenever we parse an SQL query, we should use a callback function that checks if the query starts with the words ALTER or DELETE. Such a function is a part of our Constraint Engine.

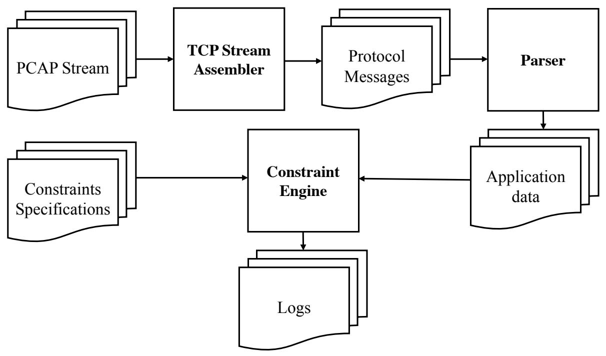


Figure 2: A High-Level Architecture of the IDS Framework.

The Constraint Engine verifies a set of constraints on the parsed application data. In case of a constraint violation, the Constraint Engine raises an alert to the user. The format of the alert does not matter to us at this point. We show this example in Figure 1.

Figure 1 shows the HTTP payload we extracted from the HTTP request in Listing 1 and how it is parsed by our targeted parser. The first command of the SQL statement triggers a callback function from our Constraint Engine. The Constraint Engine compares the string format of the SQL command to the DELETE string. In case of a match, an alert function is called which alerts the user about the usage of the DELETE query.

Thus, we can detect intrusions based on constraint violation. In this paper, we focus on the parser module and discuss it in details.

2.3 The IDS Architecture

The parser that we developed serves as a critical component of a constraint-based IDS. As depicted in Figure 2, the IDS framework has three main modules: the TCP Stream Assembler, the Parser, and the Constraint Engine.

TCP Stream Assembler. This is the first module of our system. The input for this module is a PCAP file containing the traffic of a limited network. We assemble the network data from the PCAP file into a TCP stream. We use Wireshark [13] to do this assembly before sending the TCP messages to the parser. We are developing our own TCP stream assembler using *libtins* [14], a C++ library by Fontanini et al. *libtins* provides TCP assembling capabilities that we can use in our project.

The Parser. The messages produced by the TCP Stream Assembler are sent to the parser. The parser parses those data streams and outputs the corresponding data structures for the constraint engine to use. The Parser module represents the main contribution of our work which is the focus of our next section. We leverage the ANTLR parser [3] to parse text based protocols such as HTTP [4] and SMTP [5].

Constraint Engine. The constraint engine receives two kinds of inputs: a) the parser-extracted application data, and b) a set of application-specific constraints. The ultimate goal of the constraint engine is to evaluate the parsed data against their constraints and logging any detected violations.

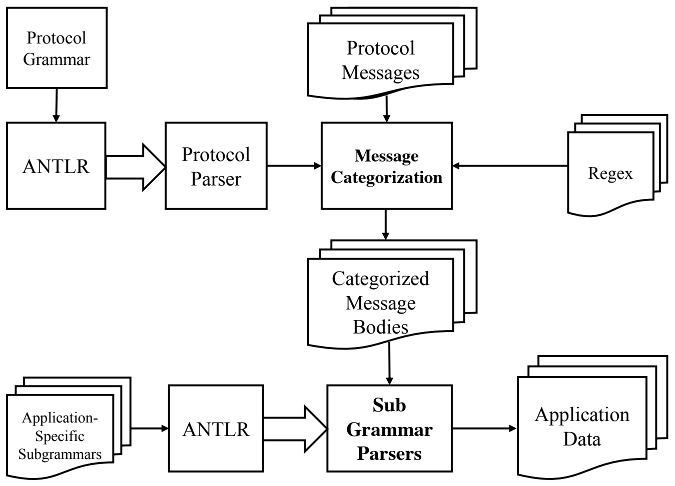


Figure 3: The Architecture of the Parsing Framework..

3 TARGETED PARSING FRAMEWORK

Figure 3 presents a high-level architecture of our framework. We use the ANTLR parser generator [3] to develop our parsing framework. ANTLR is a parser generator that can read, process, execute, and translate both binary and text inputs. It can generate parsers in *Java*, *C++*, and *C#*. Being reliable and efficient, ANTLR is widely used to build languages, tools, and frameworks in both academic and industrial applications. Twitter [15], for example, uses ANTLR to parse over 2 billion queries a day. All that ANTLR needs to generate a parser is a grammar file written in an EBNF [16] format. Our framework takes as input at least two ANTLR grammars: (1) a generic grammar, and (2) application-specific grammars (subgrammars). We can have more than one subgrammar for more than one category of messages.

The parser extracts application data from network protocol messages and sends them to the Constraint Engine. The same protocol can represent data internally in different formats. The data can be represented in JSON, XML, HTML or any application-specific format. Thus, we are unable to parse of all protocol messages with one generic grammar. Instead, we split the parser into two main phases: Message Categorization and Target-Specific Parsing. In this section we discuss these two parsing phases.

3.1 Message Categorization

To categorize a message, we first parse the message with a generic protocol parser, generated by the main grammar. This generic parser generates a parse tree out of an input protocol message. Next, we compare one or more nodes of the generated parse tree to regular expressions. A matching regular expression identifies the category of the message. If no match was found, no further processing is done on the protocol message. The message element in the general parse tree that contains the application data is extracted and labeled with the category to which it belongs. For example, if the main grammar is HTTP, we parse an HTTP message using a parser generated from a generic HTTP grammar. Using the resulting parse tree, we compare the URI node (or possibly a mime header node) to a regular expression. The matching regular expression determines the category the HTTP message belongs to. We extract the element containing the application data of that category. We end up with categorized application data that is ready to be fed into the Application-specific parsing phase.

3.2 Subgrammar Parsers

In this phase, we parse the categorized application data using a parser generated from a grammar specific to that category. We refer to the grammars in this phase as *subgrammars*. These subgrammars are accurate enough to extract all the data that we need to send to our constraint engine for evaluation. We use *ANTLR* and *C++* to generate the parsers. Figure 4 shows the message body after being parsed by the main grammar. Figure 5 shows the message body after being parsed by a target-specific subgrammar. In this phase, the parser is domain-specific which allows effective extraction of the application data that we are interested in. In our proposed framework, the parser goes through the following steps:

(1) Categorize messages;
(2) Extract the application data element;
(3) Label the extracted data with their category;
(4) Parse the labeled data with category-specific parsers.

We created a generator that generates parsers that include all the features mentioned above. We generate the parsers from a parser specifications file. We developed a parser specification language to facilitate the usage of the parser framework. The language was developed to be simple enough for the network engineers to specify their parsing needs. The language allows us to specify the following aspects of the parser: the main grammar, message categories, the regular expressions used for categorization, and any functions to apply on the parsed data.

```
 1 MainGrammar(HTTP, "HTTP") {
 2 # Main grammar scope
 3 }
 4 Category(LoginCategory) {
 5 # Categorization of LoginCategory
 6 Categorization("url",".(.?mode=login)")
 7 # Specify the application data node
 8 DataNode("message_body", "LoginHTTP")
 9 # Run functions
10 print("LoginHTTP","username")
11 print("LoginHTTP","password")
12 }
```

Listing 2: Message Category Specification.

Main Grammar Specification. To specify the main grammar, we use the keyword called `MainGrammar` which requires two arguments. The first argument is what the user chooses to call this grammar so that they can refer to it later when specifying a set of constraints for the constraint engine to use. Constraint specification is part of our future work. The second argument is the name of the *ANTLR* grammar file that specifies a particular grammar. We can define the main grammar in the specification file as shown in line 1 of Listing 2. After the first line, we declare the scope of `MainGrammar` using curly braces where we can specify different functions and attributes that refer to the main grammar.

Category Name. We specify a name for a message category. We can use the keyword `Category` followed by the category name written between parenthesis. Following that, we declare a scope for the category using curly braces. Within the scope, all specifications will refer to that category. Line 4 of Listing 2 is an example of a category called the `LoginCategory`.

Categorization Attributes. In order to categorize a message, we specify the message element that belongs to a particular category. While the main grammar defines the message elements, we also have to specify the regular expression that the parser will compare to that element. We determine the category of a message when a regular expression matches with a message element. To specify a regular expression we use the keyword `Categorization` as exemplified on line 6 of Listing 2.

Application Data Node. Another attribute to specify is the element of the protocol message that we want to parse using a specific parser. We use the keyword `DataNode` followed by two arguments that specify the name of the message element and the name of the parser's grammar (e.g., line 8 of the Listing 2).

Node Functions. The specification language allows us to apply functions to the application data discovered by the targeted parser. The purpose of these functions is to show that we have access to the application-specific data. As an example, we implemented the `print` function which prints out an element to the standard output. Two arguments follow the `print` keyword, the name of the specific grammar and the name of the element in that grammar that we wish to print on the screen. This can be seen in Listing 2 on lines 10 and 11.

The example shown in Listing 2 refers to the categorization of the Login request messages of a web application; i.e., messages that belong to the Login category. We categorize the request by matching the `url`, extracted by the main grammar parser, to the regular expression shown on line 3. We represent the regular expression in C++ standards [17]. The main grammar contains an element called `message_body` that we are interested in further parsing with a subgrammar parser. We specify the subgrammar parsing on line 8. The name `LoginHTTP` refers to the ANTLR grammar file written for this particular message category. Finally, we use the keyword `print` on the `username` and `password` message elements to display them on the screen.

4 EVALUATING THE PARSING FRAMEWORK

We generated a number of custom parsers using the parser specification language to test our framework. Each parser refers to a particular application built on top of a text-based protocol. We apply those parsers to the TCP streams generated from the application's network traffic. In this section we discuss our evaluations on two HTTP-based applications called the GraphDB and the PhpBB, and for parsing email messages sent via SMTP.

4.1 Evaluation with GraphDB

GraphDB [18] is a web implementation of a graph database that uses SPARQL [19] as its query language. The GraphDB interface communicates with its server via HTTP, sending SPARQL queries

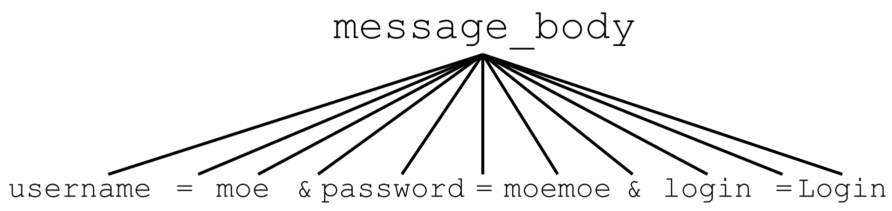


Figure 4: Message body element from the Main Grammar parse tree.

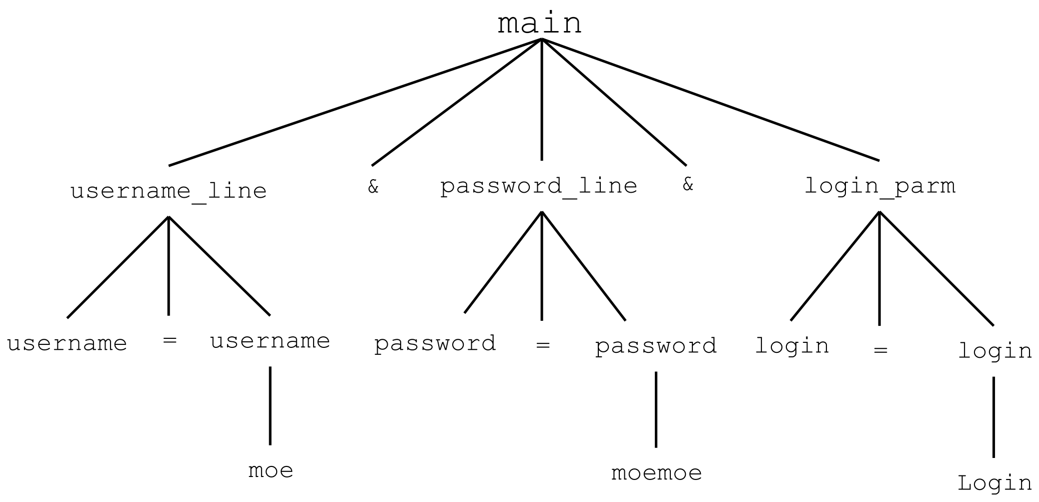


Figure 5: Message body element from the Subgrammar parse tree.

in the message body and receiving back a JSON object containing the query results.

In order to parse GraphDB's traffic, we use an HTTP grammar in ANTLR and specified it as the main grammar. This main grammar can parse HTTP headers and locate the message body. However, it is unable to parse the message body as an HTTP message body can be almost any arbitrary form or format. We wrote a SPARQL grammar as a subgrammar for parsing the SPARQL queries embedded in the HTTP requests. We also wrote a JSON grammar to handle the server's reply.

We show the specification for the GraphDB parser in Listing 3. We specified that the categorization of the GraphDB requests that contain SPARQL queries is done by matching the `Accept Info` header of HTTP to the regular expression: `".*sparql-results\\+json.*"` on line 7. On line 15, we specified that the categorization of the query results messages is done by matching the `Content Type` header to the expression: `".*sparql-results\\+json.*"`. For testing, we specified in the parser specification language to print out the predicate of the SPARQL queries and to print out the JSON values of the query results as seen on line 10 and 18.

```
1  MainGrammar(HTTP, "HTTP") {
2    # Main grammar scope
3  }
4  Category(query, "query") {
5    # Look for the SPARQL query requests
6    # to the server.
7    Categorization("accept_info", ".*sparql-
       results\\+json.*")
8    DataNode("message_body", "SparqlHTTP")
9    # testing
10   print("SparqlHTTP", "predicate")
11 }
12 Category(result, "result") {
13   # Look for the results of the SPARQL
14   # queries sent by the server
15   Categorization("content_type", ".*sparql-
       results\\+json.*")
16   DataNode("message_body", "Json")
17   # testing
18   print("Json", "json_value")
19 }
```

Listing 3: GraphDB Parser Specification.

To implement this parser, we need three ANTLR grammar files, one for the HTTP parser, one for the SPARQL parser and one for the JSON parser. The SPARQL and JSON parser will parse the payload of the HTTP grammar, which is in the `message_body` non-terminal.

We show the ANTLR definition of the message_body non-terminal below:

```
message_body: ~((HTTPVERSION1_0 |
                HTTPVERSION1_1))+;
HTTPVERSION1_0: 'HTTP/1.0';
HTTPVERSION1_1: 'HTTP/1.1';
```

The message body rule states that any character sequence that is not equal to HTTP/1.0 or HTTP/1.1 is part of the message_body non-terminal. The reason for this is that the HTTP messages are sometimes aligned right after each other. Therefore, the parser needs to keep parsing any character as a message body until we reach a new message which starts with either HTTP/1.0 or HTTP/1.1. We realize that this is not the optimal parsing rule for the message body since HTTP/1.0 or HTTP/1.1 can still appear within a message body, but for our test cases, it was sufficient. We do not show the entire HTTP grammar due to limited space. Two other grammars for SPARQL and JSON are used to parse the message_body nodes from the HTTP tree. Listing 4 shows the SPARQL grammar.

```
main: QUERY EQUAL query;
query: prefix* dataset* result pattern
    modifiers extensions;
prefix: PREFIX prefix_name COLON link;
prefix_name: ID;
link: LESSTHAN url MORETHAN;
url: http COLON FORWARDSLASH FORWARDSLASH
    (ID | DOT | ENCODING | DASH |
    UNDERSCORE | FORWARDSLASH | NUMBER)*;
http: HTTP | HTTPS;
dataset: FROM link;
result: action (ASTERISK | variables);
variables: variable+;
variable: QUESTION ID;
action: SELECT;
pattern: WHERE LBRACE condition+ RBRACE;
condition: subject predicate object DOT?;
subject: variable | entity;
predicate: variable | entity;
object: variable | entity | string;
string: DQUOTE .*? DQUOTE;
entity: ID COLON (ID | NUMBER);
modifiers: modifier;
modifier: limit;
limit: LIMIT NUMBER;
extensions: (AND extension)+;
extension: (ID | LIMIT) EQUAL (ID | NUMBER
    );
```

Listing 4: SPARQL grammar.

We parse the message body of the Query category HTTP messages with the SPARQL grammar. Messages in the Result category, are parased with the JSON grammar in Listing 5.

We installed GraphDB on a local machine and captured the traffic that involved database queries. Our generated parser was able to categorize the messages and parse the SPARQL queries. The output of our parser is shown in Listing 6. Line 2 of Listing 6 is the predicate of the SPARQL query used as printed by line 10 of the specification in Listing 3. %3A is the HTML encoding for the colon character giving the parsed predicate in the query as rdf:type. The rest of the output in Listing 6 is generated by the print statement on line 18 of the specification. This line extracts the json_value non-terminals from the JSON parse tree of the query results and prints them. Since the json_value non-terminal is recursive, some elements are printed more than once (e.g. ["s"] and "s" on lines 3, 4, 5).

```
main: json;
json: json_object | json_array;
json_object: LBRACE json_pair (COMMA
    json_pair)* RBRACE | LBRACE RBRACE;
json_pair: STRING COLON json_value;
json_array: LBRACKET json_value (COMMA
    json_value)* RBRACKET | LBRACKET
    RBRACKET;
json_value: STRING | NUMBER | json_object
    | json_array | TRUE | FALSE | NULLL;
```

Listing 5: JSON grammar.

```
 1 Start of HTTP  parser.
 2 rdf%3Atype
 3 {"vars":["s"]}
 4 ["s"]
 5 "s"
 6 "uri"
 7 "http://purl.org/dc/elements/1.1/title"
 8 {"s":{"type":"uri","value":"http://purl.
       org/dc/elements/1.1/creator"}}
 9 {"type":"uri","value":"http://purl.org/dc/
       elements/1.1/creator"}
10 "uri"
11 "http://purl.org/dc/elements/1.1/creator"
12 End of HTTP parser.
```

Listing 6: GraphDB parsing output.

4.2 Evaluation with PhpBB

PhpBB [6] is an open source bulletin board web-platform used by millions of users. It uses HTTP for client-server communication. PhpBB includes features such as posting to forums, commenting on posts, account registration, and admin management. These represent different message categories in our targeted parsing framework.

The main grammar is the same HTTP grammar used in the GraphDB test. This shows the framework is modular and reusable. Reuse of existing grammars allows focus on the application data subgrammars. We use new subgrammars to parse the PhpBB-specific application data. In this example, subgrammars for four different message categories: user login, user logout, new user registration, and file upload are used. Listing 7 shows the specification for the four message categories.

```
 1 MainGrammar(HTTP, "HTTP") {
 2   # Main Grammar Scope
 3 }
 4 Category(Login, "Login") {
 5   Categorization("uri", ".*(\\.php\\?mode=
       login)")
 6   DataNode("message_body", "LoginHTTP")
 7   print("LoginHTTP", "username")
 8   print("LoginHTTP", "password")
 9 }
10 Category(Logout, "Logout") {
11   Categorization("uri", ".*(ucp\\.php\\?
       mode=logout.*)")
12   DataNode("uri", "LogoutHTTP")
13   print("LogoutHTTP", "sid_value")
14 }
15 Category(Upload, "Upload") {
16   Categorization("uri", ".*(posting\\.php
       \\?mode=post.*)")
17   DataNode("message_body", "UploadHTTP")
18   print("UploadHTTP", "filename_value")
19 }
20 Category(Register, "Register") {
21   Categorization("uri", ".*(ucp\\.php\\?
       mode=register.*)")
22   DataNode("message_body", "RegisterHTTP")
23   print("RegisterHTTP", "email_value")
24 }
```

Listing 7: Parser Specifications for PhpBB.

On line 2 we specify that the main grammar is HTTP. Each of the four categories has its own unique categorization rule and data node. The Login category includes all the HTTP messages that a PhpBB client sends to login to the user account. Login messages are categorized by matching the URI to the regular expression `.*(\\.php\\?mode\\login)"` as specified on line 6. The message body node of a Login message is reparsed with the subgrammar `LoginHTTP` on line 7. Lines 8 and 9 print out the `username` and `password` non-terminals from the `LoginHTTP` subgrammar.

The categorization rule on line 11 of the specificaiton states that logout messages are identified by matching the URI to the regular expression `".*(ucp\\.php\\?mode=logout.*)"`. Line 13 specifies that the URI of the logout messages will be parsed with the `LogoutHTTP` subgrammar. On line 14, the values of the `sid_value` non-terminal are printed. Similarly, we specify the Upload and Register categories on lines 16 to 25. The Upload category is the category of messages that represent uploading a file to the PhpBB server. The Register category is the category of the messages that register a new user in the PhpBB application.

Listing 8 shows the `LoginHTTP` subgrammar we used for the Login category of PhpBB messages. On line 3 we define the `username` non-terminal that will contain the username that logged in. On line 5 we define the `password` non-terminal that will contain the password of the account that logged in. Those non-terminals are the ones that we refer to in our specifications file on lines 8 and 9 in Listing 7.

```
 1 main: username_line AND password_line AND
       ((login_parm AND redirect) | (redirect
        mode_param AND sid AND redirect AND
       login_parm));
 2 username_line: USERNAME EQUAL username;
 3 username: (ID | HEX);
 4 password_line: PASSWORD EQUAL password;
 5 password: TEXT | (ID | HEX);
 6 login_parm: LOGIN EQUAL login;
 7 login: (ID | HEX);
 8 redirect: REDIRECT;
 9 mode_param: MODE EQUAL LOGIN;
10 sid: SID EQUAL HEX;
```

Listing 8: LoginHTTP Subgrammar for PhpBB.

Listing 9 shows the `LogoutHTTP` subgrammar that we used for the Logout category of PhpBB messages. Here, we are interested in finding out the value of the session ID of the user that has logged out. Line 6 defines the `sid_value` non-terminal that contains the session ID of the user that has logged out. That is the same non-terminal we refer to on line 14 in Listing 7.

```
1 main: uri QUESTION variable? (AND variable
      )* AND? sid (AND variable)*;
2 uri: ~(QUESTION)+;
3 variable: ID EQUAL variable_value;
4 variable_value: value;
5 sid: SID sid_value;
6 sid_value: value;
7 value: ~(AND | EOF | NEWLINE)+;
```

Listing 9: Logout Subgrammar for PhpBB.

Similarly, we define the subgrammars for the two other message categories as shown in Listings 10 and 11. Note that some of the grammar was ommitted for the sake of brevity.

```
main: NEWLINE? username AND email AND
    password AND language AND tz_date AND
    tz AND tz_copy AND confirm_code AND
    confirm_id AND agreed AND change_lang
    AND confirm_id AND submit AND
    creation_time AND form_token NEWLINE?;
username: USERNAME username_value;
username_value: ID;
email: EMAIL email_value;
email_value: ~(AND | NEWLINE)*;
password: NEW_PASSWORD new_password_value
    AND CONFIRM_PASSWORD
    confirm_password_value;
new_password_value: password_value;
confirm_password_value: password_value;
```

```
password_value: ~(NEWLINE | AND)+;
language: LANGUAGE language_value;
language_value: ID;
tz_date: TZ_DATE utc_time PLUS MINUS PLUS
    timezone PLUS MINUS PLUS date COMMA
    PLUS a_time;
utc_time: UTC_NUMBER COLON NUMBER;
time: hours COLON minutes;
hours: NUMBER;
minutes: NUMBER;
timezone: ID FORWARDSLASH ID;
date: day PLUS month PLUS year;
agreed: AGREED bool_value;
bool_value: TRUE | FALSE;
change_language_value: NUMBER;
submit: SUBMIT ID;
creation_time: CREATION_TIME NUMBER;
form_token: FORM_TOKEN ID;
```

Listing 10: Register Subgramamr for PhpBB.

```
main: (row new_line?)* content_type file;
content_type: 'Content-Type:'
    content_type_value;
content_type_value: ID;
row: row_value? new_line? MINUS* number
    new_line content_disposition SEMICOLON
     name (SEMICOLON filename)? new_line;
filename: 'filename=' filename_value;
filename_value: STRING;
content_disposition: 'Content-Disposition:
    ' content_disposition_value ;
content_disposition_value: ID;
name: 'name=' name_value;
name_value: STRING;
number: NUMBER;
new_line: NEWLINE;
row_value: (~'\n')+;
file: .+?;
```

Listing 11: Upload Subgrammar for PhpBB.

We generated a parser from a specifications file and ran it against captured PhpBB traffic. The parser was able to recognize all four categories and printed out application-specific data such as usernames, passwords, and session IDs.

4.3 Evaluation with Email Messages

Our framework can be applied to applications built on top of other text-based protocols. We test the effectiveness of our targeted-parsing framework on email messages sent via SMTP. To collect SMTP traffic, we set up a local email server (SquirrelMail v 1.4.22) on an Ubuntu machine and created random user accounts. We started sending emails and captured the network traffic using Wireshark. We wrote a basic SMTP grammar to be our main grammar. We wrote a subgrammar that parses email messages found in SMTP traffic. To make things more interesting, we added a specific rule in the email subgrammar that parses website links. Then we specified in our parser specifications file to print out the sender and receiver of the email message and all the web links in that message. We present the SMTP main grammar in Listing 12. Listing 13 presents the Email subgrammar and Listing 14 shows the specifications for the SMTP parsing.

```
main: (message NEWLINE?)*;
message: request | reply;
reply: reply_code .*? NEWLINE;
reply_code: NUMBER;
request: email NEWLINE | ID .*? NEWLINE;
email: email_header+ .*? NEWLINE+ DOT;
email_header: ID COLON .*? NEWLINE+;
```

Listing 12: SMTP Main Grammar.

```
main: headers content;
headers: (header NEWLINE)+;
header: date | to | from | reply_to |
    subject | mime_version | content_type
    | content_transfer_encoding |
    extension_header;
extension_header: extension_name COLON (
    extension_value NEWLINE)+? header?;
extension_name: ID;
extension_value: ~(NEWLINE)+;
date: DATE COLON ~(NEWLINE)*;
to: TO COLON email | TO COLON ID LESSTHAN
    email MORETHAN;
email: (ID | NUMBER | UNDERSCORE | DOT)*
    AT host;
host: ID (DOT ID)*;
from: FROM COLON email | FROM COLON ID
    LESSTHAN email MORETHAN;
reply_to: REPLYTO COLON email;
subject: SUBJECT COLON ~(NEWLINE)*;
mime_version: MIMEVERSION COLON
    version_number;
version_number: NUMBER (DOT NUMBER)*;
content_type: CONTENTTYPE COLON ~(NEWLINE)
    *;
content_transfer_encoding:
    CONTENTTRANSFERENCODING COLON ~(
    NEWLINE)*;
content: (link | .)+? NEWLINE DOT NEWLINE
    ?;
link: (HTTP | HTTPS)? (ID | NUMBER |
    UNDERSCORE) ((DOT) (ID | NUMBER |
    UNDERSCORE))+;
```

Listing 13: Email Subgrammar.

```
MainGrammar(SMTP, "SMTP") {
  # Main Grammar Scope
}
Category(email, "email") {
  Categorization("email", "(.|\\n|\\r)*")
  DataNode("email", "EMAIL")
  print("EMAIL", "from")
  print("EMAIL", "to")
  print("EMAIL", "link")
}
```

Listing 14: SMTP Parser Specifications.

Below we show the output from our parser after we ran it on the SMTP traffic.

```
Start of SMTP parser.
From:zoe@example.com
To:moe@localhost
http://example.com
End of SMTP parser.
```

Not only does it show an extracted link, but it also provides who the message is from and who it is to. This example shows the flexibility of the parser.

4.4 Evaluation with the Constraint Engine

After we tested our generated parser, moved on to evaluating the application of our work in an intrusion detection scenario. We sent the application data parsed by our parser to a Constraint Engine prototype that we developed. We implemented two constraints in our Constraint Engine for phpBB traffic:

AC1: A login username's characters must belong to a predefined set of characters. This is an example of a single-message constraint. A single-message constraint is a constraint that the Constraint Engine evaluates by checking the contents of only one message. We show AC1 in Figure 6.

To implement AC1, we include a character verification function in the constraint engine module. This function takes as input the set of valid characters and a string to verify. To integrate this functionality in our parser, we make use of the callback functions generated by ANTLR. The callback functions are functions that are called when the parser reaches a non-terminal node. We added a callback function on the parsed username. This callback function calls the character verification module in our Constraint Engine and displays the output of the evaluation as shown in Figure 6. Our list of allowed characters includes lower-case alphabets. Therefore, the Constraint Engine reports the usage of character "3" as a violation.

AC2: A request to the server with a set session ID must only be issued from a client IP address that logged in to that session ID. We call this a multi-message constraint since the Constraint Engine needs multiple messages to evaluate it. We show AC2 in Figure 7.

AC2 is important to detect session hijacking in a local network. An attacker may extract a session ID from the network traffic between a host and a server. Then, the attacker may use the session ID to gain access to restricted information. AC2 makes sure that network requests have not been hijacked by making sure that a host sending a request with a set session ID has actually previously logged in to that session. AC2 requires the Constraint Engine to keep track of the IPs the login messages are from and the session IDs used in HTTP requests.

In AC2, we use the categorization feature of our parser to categorize AC2 related messages into three different categories: Login messages, request messages and logout messages.

Login Messages: These are the messages used to log in to an account. When the parser parses a Login message, the Constraint Engine saves the Session ID (SID) and IP pair that has logged in. We call the data structure used to save the SID-IP pair as the constraint tree.

Request Messages: These are generic messages used to browse through the phpBB website. When the parser parses a Request message, the Constraint Engine looks up the SID-IP pair of the request message to check if the pair has been previously saved by the Constraint Engine.

Logout Messages: These are messages used to log out from an account. When the parser parses a Logout message, the Constraint Engine deletes the saved SID-IP pair.

We input to our parser a Login message, a Request message with an SID that we manually modified, and a Logout message. The output of the Constraint Engine is shown in Figure 7. We received a false positive at first, saying that an SID-IP pair was not found in our constraint tree. This is because the first Login message itself has a set SID that has not been previously saved. The Constraint Engine then shows a true positive. It managed to detect that the SID-IP pair of the Request message we sent has not been previously logged in. We manually modified the SID of the Request message to test our constraint.

5 RELATED WORK

The topic of parser generation for network protocols has been discussed numerous times before. Every research done on parser generation has its own merits. Our approach depends on island grammars [20] [21]. An island grammar is a grammar that defines portions of interest (the islands) that are present within message elements that we are not interested in (the water). In comparison to our work, the islands represent the subgrammars. The water represents all the message elements we ignore. Deursen et al. and Moonen use island grammars in reverse engineering to extract elements from source code. Our work makes use of the island grammar approach to extract application data from network traffic. We provide an interface for the user to use this approach at a higher level of abstraction.

Borisov et al. [22] present a similar framework for protocol analysis. They describe a Generic Application-Level Protocol Analyzer (GAPA) with its own specifications language (GAPAL). GAPA is a powerful network analyzer that works for both binary and text-based protocols. GAPA supports layering, a feature that acts as separate instances of the GAPA engine that parses incoming packets at different levels of abstraction. Borisov et al. used layering to implement a targeted-parsing feature, similar to our framework. However, GAPA does not categorize incoming messages. GAPA sends an incoming message through its parsing layers regardless of its context. In contrast, our framework classifies incoming messages according to their elements, then parse each category with

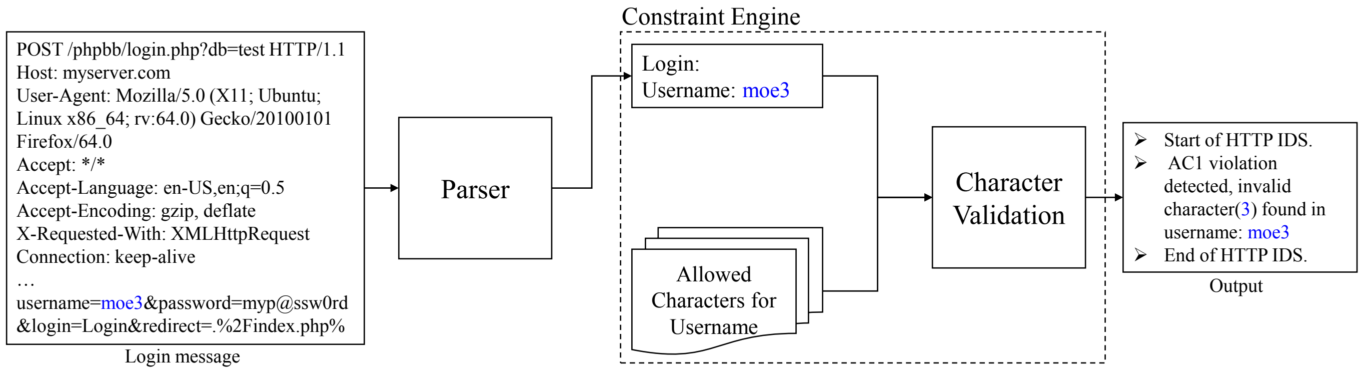

Constraint Engine
POST /phpbb/login.php?db=test HTTP/1.1
Host: myserver.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:64.0) Gecko/20100101 Firefox/64.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
X-Requested-With: XMLHttpRequest
Connection: keep-alive
…
username=moe3&password=myp@ssw0rd&login=Login&redirect=.%2Findex.php%
Login message
Parser
Login: Username: moe3
Allowed Characters for Username
Character Validation
Start of HTTP IDS.
AC1 violation detected, invalid character(3) found in username: moe3
End of HTTP IDS.
Output

Figure 6: Single Message Constraint Test.

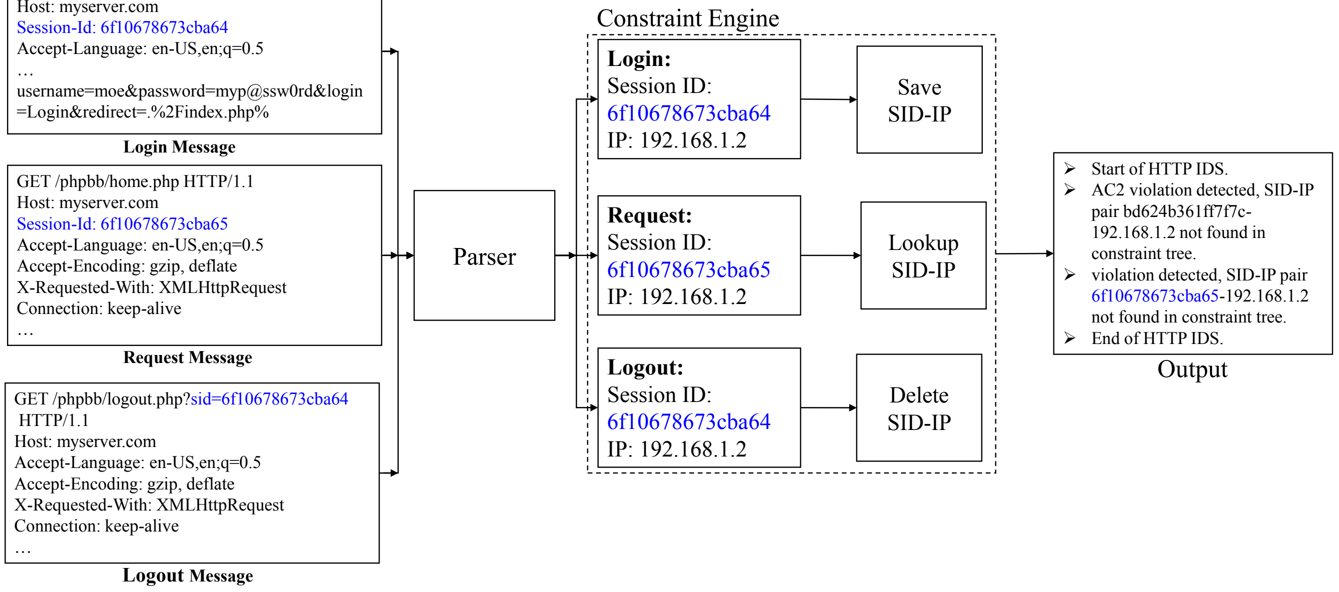

POST /phpbb/login.php?db=test HTTP/1.1
Host: myserver.com
Session-Id: 6f10678673cba64
Accept-Language: en-US,en;q=0.5
…
username=moe&password=myp@ssw0rd&login=Login&redirect=.%2Findex.php%
Login Message
GET /phpbb/home.php HTTP/1.1
Host: myserver.com
Session-Id: 6f10678673cba65
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
X-Requested-With: XMLHttpRequest
Connection: keep-alive
…
Request Message
GET /phpbb/logout.php?sid=6f10678673cba64 HTTP/1.1
Host: myserver.com
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
X-Requested-With: XMLHttpRequest
Connection: keep-alive
…
Logout Message
Parser
Constraint Engine
Login: Session ID: 6f10678673cba64 IP: 192.168.1.2
Save SID-IP
Request: Session ID: 6f10678673cba65 IP: 192.168.1.2
Lookup SID-IP
Logout: Session ID: 6f10678673cba64 IP: 192.168.1.2
Delete SID-IP
Start of HTTP IDS.
AC2 violation detected, SID-IP pair bd624b361ff7f7c-192.168.1.2 not found in constraint tree.
violation detected, SID-IP pair 6f10678673cba65-192.168.1.2 not found in constraint tree.
End of HTTP IDS.
Output

Figure 7: Multi-Message Constraint Test.

a specific parser.We use the popular and well-maintained ANTLR parser generation framework while GAPA created their own.

Burgy et al. [23] present *Zebu*, an annotated form of Augmented BNF (ABNF) [24] used to specify application data parsing of text-based protocols such as HTTP. Annotations include application-specific information such as the message element that contains application data. The Zebu approach implements application-specific parsing by modifying the ABNF grammar. In our work, we use a modular approach to keep the grammar files separate from the parser specification files. Users can reuse existing grammars to create custom parsers. Another advantage of modularity is that the parser does not require significant structural changes to update with new subgrammars.

Wondracek et al. [25] show an automatic approach to generate network protocol parsers. Their work supports parsing both binary and text-based protocols while our scope is only text-based protocols. One of the key features of the parsing framework by Wondracek et al. is that it supports auto-generation of the protocol-specific grammars instead of manually writing them. For our parsing framework, such a feature would be useful to avoid manual efforts of writing the ANTLR grammars by the domain experts. The auto-generation feature would also eliminate human-induced errors. This latter aspect is important since the accuracy of our parsing is determined by the accuracy of the input ANTLR grammars to our framework. The generated parsers by Wondracek et al. were able to achieve 100% accuracy in some protocol analyses but reached as low as 75% in others. A low parsing accuracy can lead to unexpected behavior for an IDS application.

Van den Brink et al. [26] worked on a similar approach to assess the quality of embedded SQL statements. To assess the quality of the SQL, they had to extract the SQL queries from source code first. They address two forms of embedded SQL statements: SQL statements in blocks of code and SQL statements assembled from distinct elements. Listing 15 provides an example of the latter case where the SQL statements are separated by string concatenation operations in Java.

Although our parser can distinguish the SQL statements from Java, it will only present the SQL as distinct message elements that can be reconstructed later on in C++ for example. We assume that the message part we are looking for is one continuous block, not a set of distinct elements that require assembling.

```
String query = "SELECT first_name"
 + " FROM " + employees_table
 + " WHERE salary <= " + max_salary
 + " AND salay > " + min_salary;
Connection conn = db.getConnection();
Statement st = conn.createSatement();
ResultSet rs = st.executeQuery(query);
```

Listing 15: SQL embedded in Java code.

While we used ANTLR for our parsing framework, one can use any source-to-source transformation language for the same purpose. There exist many source-to-source transformation languages that could be used instead of ANTLR. An example is TXL [27]. TXL requires two components to operate: a grammar describing what TXL will be transforming from, and, a set of transformation rules to apply to the input source. TXL is a powerful tool used in industrial and academic applications such as programming language processors, program analyzers, document processors, and many others. We chose ANTLR over TXL since TXL uses a file-based approach. The input and output of TXL are stand-alone files while ANTLR operates entirely in memory, alowing us to read directly from network interfaces in the future.

6 CONCLUSION AND FUTURE WORK

In this paper, we presented a parser generation framework that allows message categorization and targeted parsing for text-based protocols. The components of our specifications are grammars written in ANTLR with a single specification file that links them. A general ANTLR grammar is used to parse the messages, and to isolate the key part of the message that identifies the category. Once the category of the message is identified using a regular expression, the application specific part of the message is reparsed using an application specific subgrammar.

Our next step is to extend the parser specification language presented in this paper and develop a constraint specification language for the IDS. Another future work to consider is to modify the parser specification language to support nested categories for deep message inspection. That would allow us to categorize message categories into more precise subcategories which can be particularly useful when dealing with complex applications. Since the ANTLR-based parser generation engine supports modular construction of grammars, this is a straightforward extension. We also plan to implement a feature similar to the work by van den Brink et al. [26] that allows us to reconstruct a message element that is broken down into several distinct components.

While our parser supports text-based protocols, it is not yet compatible with encrypted protocols. Our target is static isolated networks in which the keys used for the network traffic are known.

REFERENCES

[1] A. ElShakankiry and T. Dean, “Context sensitive and secure parser generation for deep packet inspection of binary protocols,” in *2017 15th Annual Conference on Privacy, Security and Trust (PST)*, Aug 2017, pp. 77–7709.
[2] S. S. Thiebaut, “Real-Time Publish Subscribe (RTPS) Wire Protocol Specification,” https://datatracker.ietf.org/doc/html/draft-thiebaut-rtps-wps-00, Internet Engineering Task Force, Internet-Draft draft-thiebaut-rtps-wps-00, Feb. 2002, work in Progress [Accessed: 09-05-2019].
[3] T. Parr, *The Definitive ANTLR 4 Reference.* Pragmatic Bookshelf, 2013.
[4] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach, and T. Berners-Lee, “Hypertext transfer protocol – HTTP/1.1,” Internet Requests for Comments, RFC Editor, RFC 2616, June 1999, [Accessed: 09-05-2019].
[5] J. Postel, “Simple mail transfer protocol,” https://www.rfc-editor.org/rfc/rfc821.txt, Tech. Rep., 1982.
[6] phpBB, “Free and open source forum software,” https://www.phpbb.com/, [Accessed: 09-05-2019].
[7] M. Hasan, T. Dean, F. T. Imam, F. Garcia, S. P. Leblanc, and M. Zulkernine, “A constraint-based intrusion detection system,” in *Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems.* ACM, 2017, p. 12.
[8] S. Axelsson, “Intrusion detection systems: A survey and taxonomy,” Technical report, Tech. Rep., 2000.
[9] N. Falliere, L. O. Murchu, and E. Chien, “W32. stuxnet dossier,” *White paper, Symantec Corp., Security Response*, vol. 5, no. 6, p. 29, 2011.
[10] B. Bencsáth, G. Pék, L. Buttyán, and M. Felegyhazi, “The cousins of stuxnet: Duqu, flame, and gauss,” *Future Internet*, vol. 4, no. 4, pp. 971–1003, 2012.
[11] S. A. Gabarro, *Using PhpMyAdmin.* IEEE, 2007, [Accessed: 09-05-2019].
[12] C. J. Date and H. Darwen, *A Guide to the SQL Standard.* Addison-Wesley New York, 1987, vol. 3.
[13] A. Orebaugh, G. Ramirez, and J. Beale, *Wireshark & Ethereal Network Protocol Analyzer Toolkit.* Elsevier, 2006.
[14] M. Fontanini, “Libtins: Packet crafting and sniffing library,” 2016.
[15] H. Kwak, C. Lee, H. Park, and S. Moon, “What is twitter, a social network or a news media?” in *Proceedings of the 19th international conference on World wide web.* ACM, 2010, pp. 591–600.
[16] L. M. Garshol, “BNF and EBNF: What are they and how do they work,” *acedida pela última vez em*, vol. 16, 2003.
[17] cpluscplus.com, “Regex syntax - C++ reference,” http://www.cplusplus.com/reference/regex/, [Accessed: 09-05-2019].
[18] R. H. Güting, “GraphDB: Modeling and querying graphs in databases,” in *VLDB*, vol. 94, 1994, pp. 12–15.
[19] J. Pérez, M. Arenas, and C. Gutierrez, “Semantics and complexity of SPARQL,” *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 3, p. 16, 2009.
[20] A. Van Deursen and T. Kuipers, “Building documentation generators,” in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, Aug 1999, pp. 40–49.
[21] L. Moonen, “Generating robust parsers using island grammars,” in *Proceedings Eighth Working Conference on Reverse Engineering*, Oct 2001, pp. 13–22.
[22] N. Borisov, D. Brumley, and H. J. Wang, “Generic application-level protocol analyzer and its language.”
[23] L. Burgy, L. Reveillere, J. Lawall, and G. Muller, “Zebu: A language-based approach for network protocol message processing,” *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 575–591, July 2011.
[24] E. YD. Crocker, “Augmented BNF for Syntax Specifications: ABNF,” https://www.rfc-editor.org/rfc/rfc5234.txt, RFC Editor, RFC 5234, January 2008, [Accessed: 09-05-2019].
[25] G. Wondracek, P. M. Comparetti, C. Kruegel, E. Kirda, and S. S. S. Anna, “Automatic network protocol analysis.” in *NDSS*, vol. 8, 2008, pp. 1–14.
[26] H. v. d. Brink, R. v. d. Leek, and J. Visser, “Quality Assessment for Embedded SQL,” in *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, Sep. 2007, pp. 163–170.
[27] J. R. Cordy, “The TXL source transformation language,” *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.