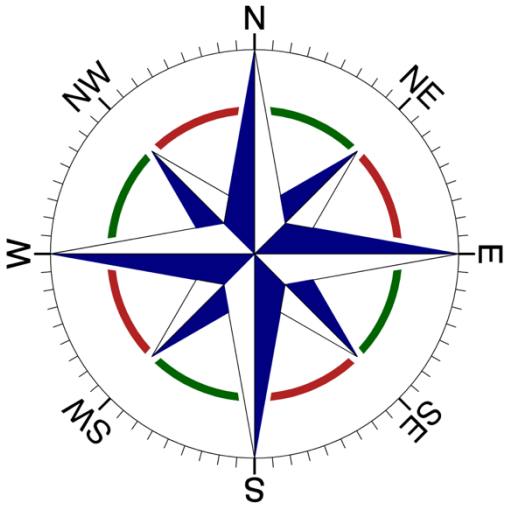
Augmenting NFS and TFTP protocols to an Intrusion Detection System

A Technical Report

By Mohammad Salloum



COPASSTR20-3
Compass Group, Queen's University
Kingston, ON, Canada
June 2020

Augmenting NFS and TFTP protocols to an Intrusion Detection System

Mohammad Salloum
Department of Electrical and Computer Engineering
Queen's University
Kingston, Canada
m.salloum@queensu.ca

***Abstract*—All computer systems are prone to malicious attacks, no matter how secure they are. There is no such thing as a fully secure system. Even isolated private networks are being targeted and successfully exploited. Therefore, network monitoring has been adopted by security engineers to detect any malicious intrusion and stop it before any damage is done. There exists an intrusion detection system (IDS) based on satisfying constraints. This IDS is practical in limited networks where the deployed network protocols are known. Currently, the IDS supports the IGMP, RTPS, ARP and NTP protocols. In this research, we will add to the parser module of the IDS the parsing of the NFS and TFTP protocols.**

***Index Terms*—NFS, TFTP, Intrusion Detection System, Network Monitoring, Parser Generator**

I. Introduction

No matter how much a system is fortified, it will always have a flaw through which an attacker can gain unauthorized access to it. Therefore, network monitoring is widely adopted between security engineers. There are manual traffic monitors who keep an eye on network traffic to find any unusual activity and stop it if they find it to be malicious. Another way to monitor a network is by using automatic intrusion detection systems (IDS) which look for exploits based on publicly exposed vulnerabilities. Such IDSs are referred to as signature-based IDSs. However, these techniques are futile against zero-day attacks. These attacks take advantage of vulnerabilities that have not been publicized yet and, therefore, the attack signature has not yet configured into the IDS. These attacks exploit flaws that can be on any level in the system's architecture, and with the thriving technological advancement, it is becoming even harder to keep track of vulnerabilities. Dean et al. [4] developed a new approach to intrusion detection. By satisfying constraints on several known protocols, the network activity can be limited to only valid traffic. Here, the valid traffic is the traffic satisfying a set of given constraints. This IDS is powerful in limited networks where all the protocols being used are known beforehand. The constrained-based IDS consists of 3 main modules: Scanner, Parser and Constraint Engine.

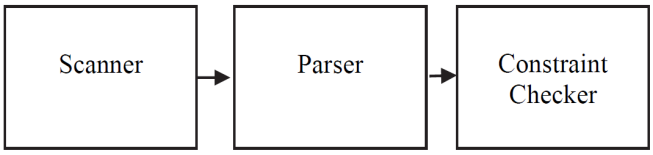


Fig. 1: Modules of the Intrusion Detection Framework

The existing constrained-based IDS currently supports four protocols (ARP, NTP, RTPS and IGMP). This limits the working scope of the IDS to networks operating with those protocols only. In this paper, we propose the adaptation of two other protocols, Network File System (NFS) [13] and Trivial File Transfer Protocol (TFTP) [16]. Our work will be specific to the generation of the parser for these new protocols.

ElShakankiry developed a parser generator [14] that accepts SCL (Structural and Context-sensitive Language) [10] definitions, a format for modeling a protocol and its constraints, to generate a secure parser in C language. Writing protocol parsers manually can be inefficient and produce low-performing or vulnerable code which may create a bottleneck in the network or introduce unwanted exploits. On the other hand, parsers which are automatically generated have high throughput and are secure. In this paper, we will develop the SCL representation of the NFS and TFTP protocol.

This paper is organized as follows: Section II goes over the background knowledge required to understand some of the paper's fundamental concepts. section III states the research plan and section IV talks about what related work is done by other researchers. Sections V and VI explain our experiment setup and implementation. Section VII specifies our research questions and what results were found for each question. We state what threats might affect the validity of our results in section VIII. At the end of the paper, we propose potential future work and a conclusion in sections IX and X respectively.

II. Background

A. SCL Language

The Structure and Context-Sensitive language (SCL), ASN.1[6] with a XML-like extension, was created to add to the ASN.1 description a more specific application-level description of a protocol. This allows a researcher to represent

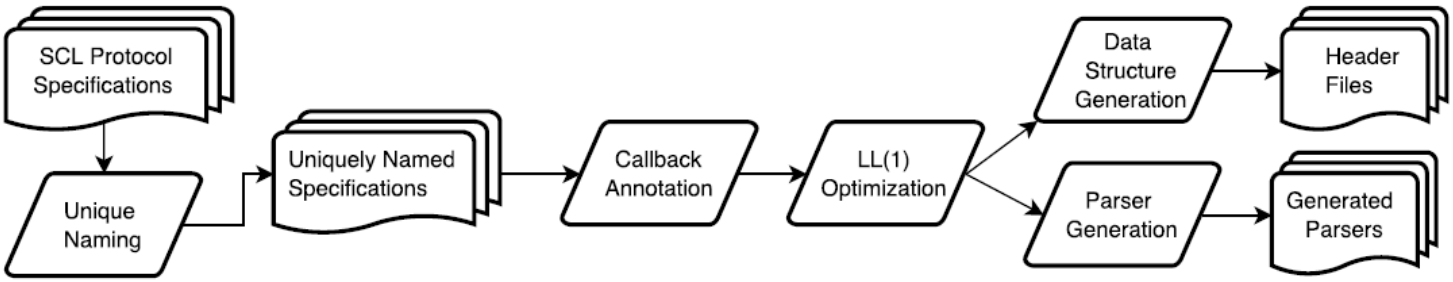


Fig. 2: Parser generation flow chart[9]

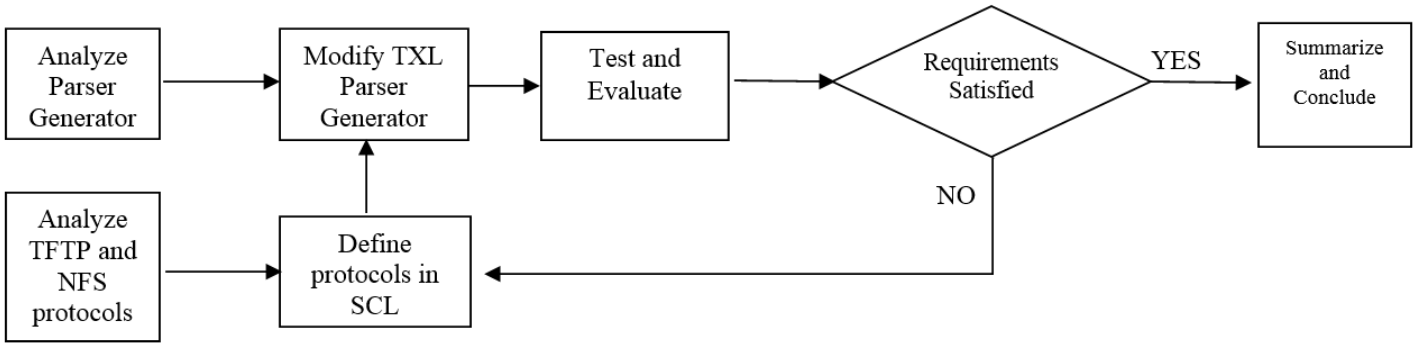


Fig. 3: Research Workflow

a protocol in a high-level abstract which is closer to the actual protocol. Thus, SCL is a great fit in our IDS system for protocol description.

B. TXL Programming Language

TXL is a powerful functional programming language used for software analysis and source transformation. It has been used for more than 15 years in industry, rapidly solving complicated computer difficulties such as the year 2000 problem [2]. A TXL software is made up of two main components, the source language description and the transformation rules. ElShakankiry et al. [14] used TXL to write the parser generator which converts SCL description into C code.

C. Protcol Parser Generation

ElShakankiry used TXL for the core of the parser generator due to its powerful source to source transformation capabilities. The input for the parser generator is the SCL description of that protocol. The SCL file is manually written through analyzing captured protocol packets. This process is described in figure 2. First, the SCL file is converted to an SCL file with added markups before the actual generation of the parser. The new SCL file also has globally unique attribute naming which is done by appending the name of the protocol to the beginning of the names. After markup, the SCL is transformed to the C code of the parser. Two C files are generated for each SCL file, one is a header file and the other is C source code. The former is used to hold C structs to be used by the generator while the latter is the actual parser.

D. NFS (Network File System)

The network file system protocol[13], developed by Sun Microsystems, allows different machines to remotely share files across networks. This protocol has cross-platform support due to the fact that it is built over the Remote Procedure Call protocol (RPC)[17] which highly supports portability. The latest version of NFS is version 4, although it is not the most popular. The most popular versions of the protocol are 2 and 3. Version 2 and 3 of NFS use UDP as a transport protocol while version 4 uses TCP as a transport protocol. We mainly focus on the latest version of NFS in our work.

E. TFTP (Trivial File Transfer Protocol)

TFTP [16] is basic protocol used to transfer files between a server and a client. One of the primary uses of TFTP was in booting a computer in a network where the operating system is stored on a separate file sharing server. This protocol is implemented on top of the UDP protocol. TFTP has only 2 features which are read from and write to files on a server. Due to its simple implementation it lacks many advanced features like list, delete or rename.

III. Research Plan

Figure 3 summarizes our approach. A lot of work has been done on the IDS and it is very time consuming to analyze the whole system. Therefore, the proposed approach was to have a general understanding of the whole system and a thorough understanding of the parts of the system that are specific to the parser generator.

A. Analyze and understand current parser generator module

To be able to add a new feature to the current system, first we have to fully understand how the system is running and study the language it was written in. In this case, the protocol structure is defined in SCL and the parser generator is written in TXL [2].

B. Analyze and study the NFS and TFTP protocols

The SCL must represent the protocols effectively, therefore, the protocols should be thoroughly studied and understood first. Reference to the protocol documentation will be made. However, the main resource for our description will be actual captured packets. Using Wireshark, the captured NFS and TFTP packet structures will be analyzed. This is important because usually the implementation of a protocol in industry is different from what the documentation states. Any missing protocol attributes will be discovered in the validation phase.

C. Designing and implementing the new parser

Now we are ready to design the new SCL structure and modify the TXL parser generator. The new parser should be integrated into the system for the work to be considered productive. Plus the final product must achieve multiple goals which are discussed further in the next step. Figure 4 shows a snippet from the SCL description of NFS which will be used to generate the parser. Note that we included a minor sample of the code because it is too lengthy to be included entirely.

D. Evaluating and Testing

The current parser generator is robust and secure and it should remain this way after the addition of the 2 new protocols to the parser generator. Syntax and semantic errors are expected in this phase. Syntax errors will appear in the generation of the parser, the parser will not be generated as long as the syntax errors exist in our code. As for semantic errors, they can be identified by testing out the parser on real NFS and TFTP packets. The new parser is acceptable if it achieves the following standards, these standards were defined by ElShakankiry et al. [14]

1) Performance: The generated parser must maintain an average processing speed of 1 GBits per second which is the average speed of intranets. In addition, speed is a crucial aspect here since the faster the IDS operates, the more damage can be avoided in case of an intrusion.

2) Reliability: Since the parser is automatically generated, it contains all required memory checks. In addition, we must ensure that the parser does not produce any false results in any possible situation. This can be validated through thorough testing of the parser on captured packets. Two well-known metrics will be used to evaluate how reliable our new parser is, these are precision and recall:

$$precision = \frac{RelevantDetectedPackets}{DetectedPackets} * 100$$

$$recall = \frac{RelevantDetectedPackets}{RelevantPackets} * 100$$

```
1 fattr ::= SEQUENCE {
2       type    ftype (SIZE DEFINED),
3       mode    INTEGER (SIZE 2 BYTES),
4       nlink   INTEGER (SIZE 2 BYTES),
5       uid     INTEGER (SIZE 2 BYTES),
6       gid     INTEGER (SIZE 2 BYTES),
7       size    INTEGER (SIZE 2 BYTES),
8       blocksize INTEGER (SIZE 2 BYTES),
9       rdev    INTEGER (SIZE 2 BYTES),
10      blocks  INTEGER (SIZE 2 BYTES),
11      fsid    INTEGER (SIZE 2 BYTES),
12      fileid  INTEGER (SIZE 2 BYTES),
13      atime   timeval (SIZE DEFINED),
14      mtime   timeval (SIZE DEFINED),
15      ctime   timeval (SIZE DEFINED)
16 }
17
18 sattr ::= SEQUENCE {
19      mode    INTEGER (SIZE 2 BYTES),
20      uid     INTEGER (SIZE 2 BYTES),
21      gid     INTEGER (SIZE 2 BYTES),
22      size    INTEGER (SIZE 2 BYTES),
23      atime   timeval (SIZE DEFINED),
24      mtime   timeval (SIZE DEFINED)
25 }
26
27 attrstat ::= SEQUENCE {
28      attributes      fattr (SIZE DEFINED)
29 }
30
31 diropargs ::= SEQUENCE {
32      dir     fhandle (SIZE DEFINED),
33      name    OCTET STRING (SIZE 255 BYTES)
34 }
35
36 diropres ::= SEQUENCE {
37      file            fhandle (SIZE DEFINED),
38      attributes      fattr (SIZE DEFINED)
39 }
```

Fig. 4: NFS description in SCL snippet

3) Security: Modifications of the current parser generator may introduce new security threats on the system. We must verify that our alterations are secure and abide to healthy security practices.

4) Usability: The original parser is modular and easy to operate and. We have to maintain these attributes to not disrupt the workflow of the original system.

The parser generator will be tested with pre-captured packets involving TFTP and NFS protocols. False positives and detection failures during testing will be indications to modify the definition of the protocols in the SCL structure.

IV. Related Work

Many researchers adopted the fields of protocol parsing and intrusion detection. We here discuss some of the papers that we found to be relevant to our work.

A. Building High-performance Application Protocol Parsers on Multi-core Architectures

Protocol parsing is an essential step in analyzing network traffic, and with the never-ending avalanche of new applications and services, more throughput is required by these parsers to not slow down networks. This paper[21] discusses a way to make use of parallel processing in CPUs to increase the speed of network parsers. As a case study, the paper uses source-to-souce transformation to transform parser code generated by FLEX into code that supports parallel processing by exploiting lock-free design principles. Our work consists of source-to-source transformation and parser generation but this paper goes an extra step to provide multi-core parser processing. This significantly increases the parsing speed. According to the paper, they were able to achieve 20 Gbps parsing speed on average HTTP packets which is 20 times faster than our parser.

B. Deep Grammar Optimization for Submessage Structure of Network Protocol Parsers

Lavorato et al. [9] introduced a new method to optimize the parsing of the traffic of the industry standard RTPS network protocol. A RTPS message is composed of a RTPS header with a set of sub-messages. Each sub-message has a distinguishable type which needs to be parsed in a specific way. Initially, the generated parser iterated through all of the RTPS message types to match the sub-message type with its corresponding identification and then handle the sub-message from there. Kyle proposed a different approach, he added a unique markup in the SCL definition of the protocol to identify the sub-message type so that each sub-message type has its own distinct callback function. This increased the overall speed of the parser by 7.66% and the full IDS system by 5.60%. It is similar to our work as it is working on improving the system they were focused on optimization of current features while our work will introduce a new feature.

C. Development and Testing of an Intrusion Detection System for Unmanned Aerial Systems

This paper[12] proposes a new intrusion detection system which is used in critical systems such as unmanned aerial systems and unmanned aerial vehicles. The unique aspect in this system is that it uses automated training based on the blackboard architecture[5] to develop its data set and based on this data, it can distinguish legitimate packets from tampered ones. The use of an AI system can be very efficient, especially with the rapid development of new application protocols. Many new protocols are being introduced every other week and it can be overwhelming to manually code them into intrusion detection systems, therefore, automating this can be very helpful. Another benefit is that this method minimizes errors since it reduces the human factor. At the same time, we run a security risk as attackers will try to find defects in the training algorithm and exploit them. In our work, our parsers are automatically generated by TXL, but the input is a manually written SCL file which represents the structure of a designated protocol.

D. Privacy preservation intrusion detection technique for SCADA systems

Supervisory control and data acquisition (SCADA) systems monitor and control critical industrial infrastructure utilities such as gas, traffic and electricity. This paper [8] discusses a new intrusion detection system using the correlation coefficient EM clustering techniques. This technique classifies SCADA data into normal and abnormal activities. The proposed IDS here is to be used in critical systems which is the case of our IDS. However, their work is mainly focused on the SCADA systems while our IDS can be used in any network environment. Another difference is that the SCADA IDS categorizes intrusions based on previous training data while our IDS works on preset rules and criteria to evaluate network activities.

V. Experiment Setup

Writing protocol descriptions in SCL is a trial and error process that requires both the protocol's documentation and a sample packet capture of that protocol. We will focus more on the latter in our work since the actual implementation of the protocol differs from its initial documentation as vendors implementing the protocol tend to modify it in a way to meet their requirements and this can be revealed only by inspecting the packets generated by the protocol.

Protocol documentations were found online[13][16]. For the NFS and TFTP traffic, the only resources found online were 300 packets of NFS and 200 packets of the TFTP protocol. These resources allowed us to start developing the structures of the protocols in SCL but they were not enough for our research. To solve this, we had to implement the protocols in a controlled environment to allow custom generation of the protocol traffic.

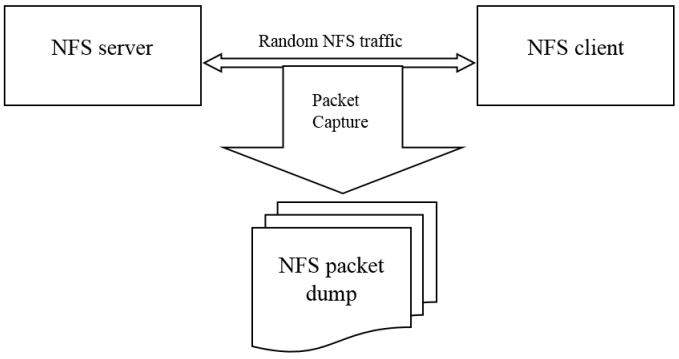


Fig. 7: NFS traffic generation setup

Our setup is represented in figure 7. An NFS server was setup on a machine running CentOS 6.9 [19]. We connected an NFS client, running Ubuntu 17.10[20], to the server and wrote a script using the bash[3] language to simulate NFS traffic. Finally, tcpdump[18] was used on the Ubuntu machine to capture these packets. These captured packets were used to progress the development of our SCL implementation of NFS.

```
RPCCALL ::= SEQUENCE {
    fragmentheader  INTEGER (SIZE 4 BYTES),
    XID             INTEGER (SIZE 4 BYTES),
    messageType     INTEGER (SIZE 4 BYTES),
    rpcVersion      INTEGER (SIZE 4 BYTES),
    program         INTEGER (SIZE 4 BYTES),
    programVersion  INTEGER (SIZE 4 BYTES),
    procedureName   INTEGER (SIZE 4 BYTES),
    credenitails    longidentifier (SIZE DEFINED),
    verifier        identifier  (SIZE DEFINED),
}
<transfer>
    Back {messageType == 0}
</transfer>
```

```
RPCREPLY ::= SEQUENCE {
    fragmentheader  INTEGER (SIZE 4 BYTES),
    XID             INTEGER (SIZE 4 BYTES),
    messageType     INTEGER (SIZE 4 BYTES),
    replystate      INTEGER (SIZE 4 BYTES),
    verifier        identifier  (SIZE DEFINED),
    acceptState     INTEGER (SIZE 4 BYTES),
}
<transfer>
    Back {messageType == 1}
</transfer>
```

Fig. 5: Side by side comparison of RPC implementations in SCL

As for TFTP, we were not able to use the same system as the Ubuntu machine was not able to establish a TFTP connection to the TFTP server running on our CentOS. To save time, we modified our environment by replacing the Ubuntu machine with a machine running Windows 7 Professional [11]. On the windows machine, we ran a TFTP server by Solarwinds [15] and we connected to this server from our CentOS machine. The connection was successful and we started transferring large files to and from the TFTP server while also capturing the network traffic using tcpdump. The captured packets were used to further progress our SCL implementations.

As mentioned earlier, the SCL implementations of the protocols are the input files for the TXL parser generator. This parser generator will generate two C files for each protocol. The first generated file is a header file holding the data structure of the protocol and the second generated file contains the main parsing functions. After generating these 2 files, they are copied to the folder which contains the rest of the source files of the IDS system to be compiled. The system is compiled as a whole and the system's parser, integrated with the new protocols, is generated. This parser is then tested against our generated traffic of each protocol and according to the results, we modify our initial SCL files. This process is repeated until the required accuracy is achieved. To speed up this process, we wrote a bash script that executes the TXL script to generate the C code parsers, copies these parsers to the rest of the project and compiles it then runs the final IDS parser against our testing packets and displays the results.

VI. Implementation

The NFS protocol has a complex structure since it is built on top of another protocol, the remote procedure protocol (RPC) [17]. RPC is an application-layer protocol that allows a process to request a service from another process running on a different machine. It operates synchronously between a client and a server, therefore, for every remote call there must be a reply. To understand the NFS protocol we must fully understand RPC first. A side by side comparison of the two RPC phases (RPC

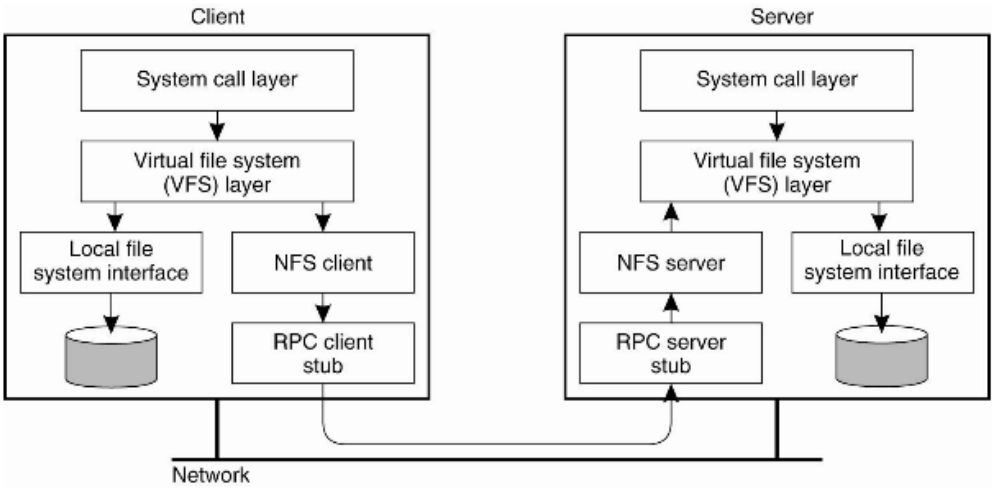


Fig. 6: NFS architecture [1]

CALL AND RPC REPLY) implemented in SCL is shown in figure 5

We will first address the CALL phase of RPC. In this phase, the structure of the protocol consists of 9 different attributes of varying memory sizes but most of them are 4 Bytes. The most important attributes to understand are the MessageType and the ProcedureName attributes. With both being the size of 4 Bytes, the former specifies whether the RPC message is a CALL or a REPLY. A value of 0 for MessageType specifies that this is a CALL message while a value 1 specifies a REPLY. As for the ProcedureName attribute, its value specifies what NFS procedure should be executed on the remote host. The protocol defines 40 different procedures, however, we are only interested in the ones that are adopted by NFS implementations which will be revealed in our packet capture. In our SCL implementation, we must specify a name and the size of each attribute and any rules that should be associated with this structure. For instance, for RPC-CALL, the attribute MessageType should always be equal to 0.

As for the REPLY phase of RPC, we have 6 attributes and most of which are the same from RPC-CALL. Both RPC-CALL and RPC-REPLY have the MessageType attribute which we discussed earlier, and the XID attribute which serves as an identification for the RPC message. A difference here would be that the MessageType must be equal to 1 for this to be a RPC reply.

An abstract NFS architecture is represented in figure 6. A NFS packet structure consists of 5 main fields, 2 of which are of interest to us: The RPC field and the OPERATIONS field. The RPC field is represented by a variable either of type RPCCALL or RPCREPLY, both of which were described earlier. As for the operations field, this defines a set of NFS operations that each NFS packet is associated with. We found 9 different operations implemented for NFS (version 4) and each with a call and a reply state, which makes a total of 18 operations. We write each operation's structure in SCL and associate it with a parent NFS packet.

TFTP has a less complex structure. The SCL code of NFS consisted of 563 lines of code while the TFTP needed only 38 lines. There are 3 main message types of TFTP which are the REQUEST, ACKNOWLEDGEMENT and DATA. The REQUEST is the message that initiates a TFTP communication while an ACKNOWLEDGEMENT message helps confirm the receive of a TFTP message. As for the DATA messages they transfer the DATA to or from a TFTP server with each packet having a maximum of 512 Bytes of payload.

The implementation of the NFS protocol in SCL was more complex than the TFTP protocol. Testing methods and results are described later in the paper.

VII. Research Questions

To guarantee a successful research, we have to set some questions that should be answered by the end of the research. Any answer, whether positive or negative, is acceptable.

A. Is our new parser efficient enough to be used in real-time networks?

The new parser generated must keep up with the average bandwidth of the initial protocol parsers (ARP, RTPS, NTP and IGMP) which is around 1 GBits per second. This is a necessity to avoid creating a bottleneck in the system. In addition, limited private networks have an average speed of 1 GBits per second so this speed must be achieved for the system to be practical in a real-time environment. The speed will be measured on the processing of pre-captured network traffic files which represent the protocols described in the paper. According to ElShakankiry, his parser generator generated parsers with an average speed of 1185.78 MBits per second [14] as shown in Table 1. Thus, our new parser is expected to be in that speed range which satisfies our speed requirement.

To test this, we used our simulation environment described earlier to generate enough traffic for our tests. We generated a traffic dump of 2313081 NFS packets (1,000 Megabytes) and a TFTP dump of 4956047 packets (1,610 Megabytes). We ran the parser against each dump 10 times while measuring the time taken each time. At the end, we computed the average time taken by the parsers. All the tests were done on a machine running on an Intel Core i7-3610QM CPU processing at 2.30 GHz. The results are show in table I and table II.

	Time taken in seconds	
	NFS Parser	TFTP Parser
Test 1	10.74	2.279
Test 2	3.666	8.037
Test 3	8.892	2.274
Test 4	8.929	1.802
Test 5	9.05	1.257
Test 6	8.892	8.265
Test 7	9.042	1.384
Test 8	1.852	2.738
Test 9	4.783	8.389
Test 10	1.847	1.311
Average	**6.769**	**3.774**

TABLE I: Time taken by parsers

	NFS Parser	TFTP Parser
Size of traffic parsed	1000906395 Bytes	1610700997 Bytes
Average time take	6.769 s	3.774 s
Average speed	**1.183 Gbps**	**3.414 Gbps**

TABLE II: Average speeds of parsers

First thing we noticed was the large variation in the time taken by the parsers to complete parsing of the given traffic. This is because of caching, the program is already in the cache of the CPU when we run it a second and a third time and therefore it will run much faster. To handle this, we rebooted our system multiple times during testing and then calculated the average of the total time taken by the parser. To verify that the large variation is because of memory caching, we investigated the correlation between the time taken by the parser to complete its processing and the size of cached memory. We used the program *free* to monitor the

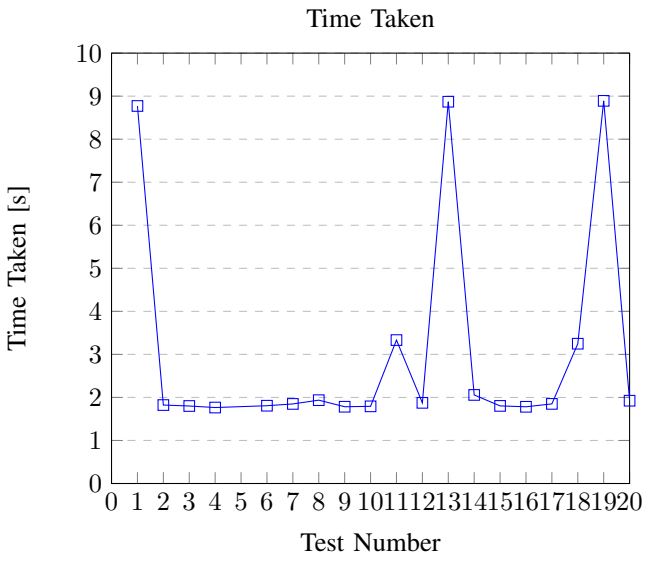


Fig. 8: Time taken for each test

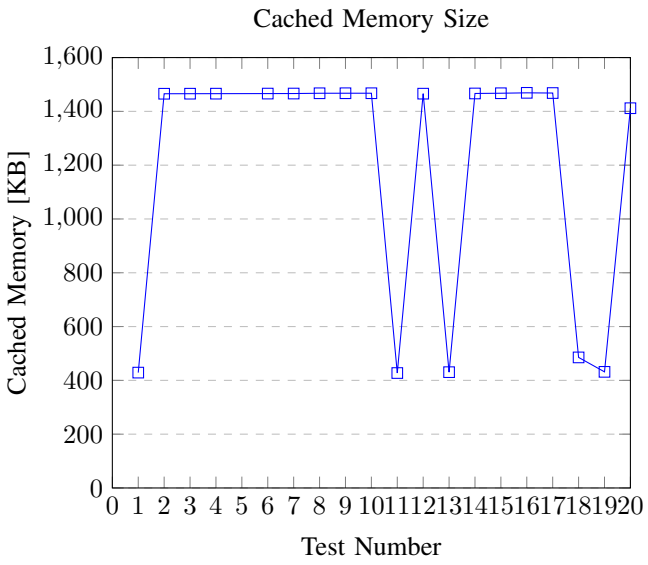


Fig. 9: Size of memory cached for each test

cached memory while we did more speed tests. The results are presented in the figures 8 and 9.

It is evident that the cache size and the speed of parsing are inversely proportional. In the first test, we see that the parsing time is 8.7 seconds and the cache size is 428 KB. Then there is a sharp decrease in parsing time in test 2, when at the same time, we witness a sharp increase in cache size. This 1.8 seconds parsing time with a cache of 1,460 KB is maintained from test 2 until test 10. After test 10, we did a soft reboot of the system to see if there is a change in readings. We notice an increase in parsing time (1.8s to 3.33s) and a large decrease in cache size (1,460 KB to 427 KB). The test right after that (Test 12) shows that the parsing time is back down to 1.8s when the cache goes up to 1,460 KB again. After that we did a hard reboot (A full shutdown before powering up again) and we got a parsing time of 8.8 seconds with a cache of 430 KB. More reboots were done in later tests and the same pattern was observed. This shows us that the cached memory is highly affecting the speed of our parser.

The average speed of the NFS parser was 1.183 Gbps while the average speed of the TFTP parser was 3.414 Gbps. This variation in the speed of the protocols is expected as the structure of the NFS protocol is more complex than the structure of the TFTP protocol. As mentioned earlier, the SCL description of the NFS structure was 563 lines long while the TFTP one was only 38 lines.

The results for efficiency met with our requirements of 1 Gbps and thus we consider that the parsers are efficient enough to be used locally in limited networks.

B. How reliable is our new parser?

For the work to be determined successful, parsers should have at least 95% precision and recall. Anything lower than 95% will be considered too low because the system is an IDS and it is meant to be used in critical scenarios where errors are not tolerable.

We calculate the accuracy and recall using the equations mentioned earlier in the paper. Our results are displayed in table III.

	NFS Parser	TFTP Parser
Total Packets	2313081	4956047
Detected Packets	2110746	4956024
Relevant Packets	2109023	4956047
Relevant Detected Packets	2109023	4956024
Accuracy	**99.92%**	**100%**
Recall	**100%**	**99.99%**

TABLE III: Precision and Recall of our parsers

These results show that our protocol description was precise. We were able to achieve 99.92% accuracy and 100% recall for the NFS parser and as for the TFTP parser we got 100% accuracy and 99.99% recall. These results meet our initially stated requirements and therefore are determined to be successful.

C. How does our newly integrated protocols affect the parsing of the other protocols?

Our ultimate goal is to improve the overall system, therefore we do not want our integration to negatively affect the performance of the protocol parsing, especially when all the protocol parsers are running coherently. The speed of parsing will be measured before our protocol integration and after. A maximum of 5% speed decrease is tolerable, a result above this limit will indicate that our integration had a negative impact on the system.

This test was carried out on RTPS. The time required to parse the traffic samples and the accuracy of the parsing was

the same before and after the addition of our new protocols. The results of this test when done on parsing 11,612 RTPS packets are displayed in table IV. To measure accuracy, we calculated what percentage of the packets provided to the parser were parsed. As for the processing time, we coded in a built-in timer in the parser and it showed us the processing time for each test. The parsing time taken was measured 10 times before and 10 times after augmenting the NFS and TFTP protocols.

	Time taken in seconds	
	Before Augmenting	After Augmenting
Test 1	0.0276	0.0138
Test 2	0.0257	0.0241
Test 3	0.0296	0.0291
Test 4	0.0285	0.0264
Test 5	0.0282	0.0289
Test 6	0.0228	0.0237
Test 7	0.0291	0.0278
Test 8	0.0241	0.030
Test 9	0.0263	0.0277
Test 10	0.0256	0.030
Average	**0.0267**	**0.0261**

TABLE IV: Parsing time test on RTPS packets

The average change in processing time after adding our new protocols was -0.0224%, which indicates a very minor speed increase that can be considered benign. This is definitely a successful outcome of our work as the initial performance was not negatively affected.

VIII. Threats to Validity

Any research is prone to errors and validity threats, and our work is no different. There are certain issues that may invalidate some of our findings.

The NFS protocol is very complex, and thus, different implementations may slightly vary parts of the protocol's structure. In addition, the simulation that we deployed may be biased and not include all the possible NFS procedures and forms which, in turn, can affect our parser's accuracy test. To solve this, our data set should contain samples from different sources to increase our scope. The same reasoning can be applied to the TFTP protocol.

During testing the speed of the parser, the results were highly affected by caching. We tried to minimize this by rebooting our system between tests. This method does not fully diminish our problem, especially since we are running these machines in virtual environments. A better approach would be to run a cache clearing program between every two tests.

IX. Future Work

Our work is the first step towards fully augmenting the NSF and TFTP protocols in the IDS. We still need to edit the constraint engine to include the constraints related to these protocols to fully complete the integration of these protocols in the system.

New methods and optimization techniques must be developed to further increase the efficiency of protocol parsing. The speed is acceptable right now but in the near future it will be too slow and it will not be able to keep up with the network speed and thus it will cause a bottleneck in the network. Our consideration should not be limited to the parsing speed as there is also the speed of evaluation of the packets that should be accounted for. A solution for this can be implementing parallel processing in our code, this was proven to be highly effective. Parallel processing in parsers was able to achieve speeds of 20 Gbps [21].

A new way can be developed to improve the addition of new protocols. We can use artificial intelligence algorithms to fully automate augmenting new parsers using sets of training data[12]. This will improve the system's efficiency and help it adapt to the rapidly changing technologies. However, such methods need to be thoroughly studied before implementing in the system to ensure we do not introduce new security risks.

X. Conclusion

In this paper we introduce two new protocols to the constrained-based IDS. It describes the effort required to integrate new protocols to the parser generation module of the IDS. More work needs to be done in order for the protocols to be fully integrated in the IDS. A lot of work went into the system and to introduce new protocol adaptations we have to work module by module and maintain the overall system stability. The IDS has been proven effective in limited networks and has been field-tested in an air traffic control system where the protocols used are restricted to the ones mentioned in this paper[7] .

One thing to note is that the IDS has decent documentation that can be further improved. Augmenting the two new protocols very time consuming. Re-factoring and further documenting the code will allow faster progress for future developers.

References

[1] San Bernardino (CSUSB) California State University. Nfs architecture. http://cse.csusb.edu/tongyu/courses/cs660/notes/dresource.php.

[2] James R. Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190 – 210, 2006. Special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA '04).

[3] GNU. Bash - gnu project - free software foundation. https://www.gnu.org/software/bash/.

[4] M. S. Hasan, A. ElShakankiry, T. Dean, and M. Zulkernine. Intrusion detection in a private network by satisfying constraints. In *2016 14th Annual Conference on Privacy, Security and Trust (PST)*, pages 623–628, Dec 2016.

[5] Barbara Hayes-Roth. A blackboard architecture for control. *Artif. Intell.*, 26(3):251–321, August 1985.

[6] ITU. Abstract syntax notation one (asn.1): Specification of basic notation., August 2015.

[7] M. Janicki and K. Strzecha. Comparation of java-dedicated parser generators. In *Mona Proceedings of the International Conference Modern Problems of Radio Engineering, Telecommunications and Computer Science, 2004.*, pages 422–425, Feb 2004.

[8] M. Keshk, N. Moustafa, E. Sitnikova, and G. Creech. Privacy preservation intrusion detection technique for scada systems. In *2017 Military Communications and Information Systems Conference (MilCIS)*, pages 1–6, Nov 2017.

[9] K. Lavorato and T. R. Dean. Deep grammar optimization for submessage structure of network protocol parser. 2017.

[10] 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*, pages 155–164, October 2005.

[11] Microsoft. Windows 7. https://www.microsoft.com/en-us/software-download/windows7.

[12] Robert Mitchell and Ing-Ray Chen. Specification based intrusion detection for unmanned aircraft systems. In *Proceedings of the First ACM MobiHoc Workshop on Airborne Networks and Communications*, Airborne '12, pages 31–36, New York, NY, USA, 2012. ACM.

[13] B. Nowicki. Nfs: Network file system protocol specification. RFC 1094, DOI, https://www.rfc-editor.org/info/rfc1094, March 1989.

[14] Ali El Shakankiry. Context sensitive and secure parser generation for deep packet inspection of binary protocols, August 2017.

[15] Solarwinds. Free tftp server. https://www.solarwinds.com/free-tools/free-tftp-server.

[16] K. Sollins. The tftp protocol (revision 2). RFC 1350, DOI, https://www.rfceditor.org/info/rfc1350, July 1992.

[17] Inc. Sun Microsystems. Remote procedure call protocol specification. RFC 1057, DOI, https://tools.ietf.org/html/rfc1057, June 1988.

[18] TCPDUMP/LIBCAP. Tcpdump/libpcap public repository. http://www.tcpdump.org/.

[19] CentOS Team. Centos (community enterprise operating system). https://www.centos.org/.

[20] Ubuntu Team. The leading operating system for pcs, iot devices, servers and the cloud — ubuntu. https://www.ubuntu.com/.

[21] K. Zhang, J. Wang, B. Hua, and X. Tang. Building high-performance application protocol parsers on multi-core architectures. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 188–195, Dec 2011.