Parsing, Performance, and Pareto in Data Stream Security

J. Peter Brady and Sean W. Smith
Department of Computer Science
Dartmouth College
Hanover, New Hampshire 03755
Email: {jpb, sws}@cs.dartmouth.edu

2024 IEEE Security and Privacy Workshops (SPW) | 979-8-3503-5487-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/SPW63631.2024.00025

***Abstract*—Adding in-line LangSec filtering to network data streams can improve security (e.g., by protecting the receiving end from crafted input attacks) but can lead to considerable performance overhead. This paper presents our GUARDS (Global Unified Approach for Reliable Data Stream Security) approach to balance between system performance and the effectiveness of in-line filtering, by allowing dynamic prioritization of security measures in high-risk regions. Our research shows that this model and DFDL parsing can effectively validate and secure the Network File System version 4 (NFSv4) protocol, achieving a balance between parsing efficiency and data integrity. This contribution helps improve the ability of network communications to withstand and maintain functionality in response to changing data representation difficulties.**

I. Introduction

Balance between performance and security is crucial in the dynamic landscape of data representation and processing. Performance demands rapid system operations, whereas robust security measures often entail comprehensive parsing and verification, potentially impeding speed. This dichotomy is particularly vexing in network communications, where speed may be critical and adding parsing to what's typically passive communcation may be seen as an unnecessary extra.

The Data Format Description Language (DFDL) (e.g. [1], [2]) offers a schema-driven approach that enhances data description and facilitates the creation of adaptable and distributable data format descriptions, and related software for comprehensive data access. DFDL enables parsers to dissect data into identifiable segments for subsequent analysis by defining a set of rules for data stream formats. Its declarative nature allows for the accurate interpretation and manipulation of any data format.

Our research focuses on leveraging DFDL to flexibly improve security of data stream systems without sacrificing performance. We develop a DFDL schema specifically for validating network protocols, emphasizing the Network File System version 4 (NFSv4) protocol [3]. As a first step, our paper assesses the effectiveness of DFDL-generated parsers in verifying complex network protocols, which is crucial for safeguarding data integrity and security in network communications.

Dans ses écrits, un sage Italien
Dit que le mieux est l'ennemi du bien.[1]
–Voltaire, La Bégueule [4]

As Section IV-A discusses, complete LangSec mediation incurs significant overhead. To mitigate the performance costs associated with improved security measures, we introduce GUARDS (Global Unified Approach for Reliable Data Stream Security). This framework incorporates novel mediation techniques, including Pareto calculations and signaling game theory, to optimize the performance of our DFDL parser by identifying and prioritizing network procedures that are most susceptible to failure or corruption; rather than trying to find a difficult-to-reach "best", we can look for a point where we have good performance and security goals.

By focusing on the most significant vulnerabilities that could lead to parsing failures, GUARDS allows efficient resource allocation to address the most critical risks, thus ensuring a harmonious balance between performance and security. This methodology improves the practical application of DFDL parsers in software systems and demonstrates a strategic approach to maintaining optimal operational efficiency and resilience in the face of security challenges.

Section II provides background information, while Section III details our methodology for solving the problems. Section IV describes our results, Section V discusses these results and describes future work, and Section VI provides the conclusion.

II. Background

A. LangSec

Language-Theoretic Security (LangSec) (e.g., [5]) is a multidisciplinary field that intersects computer science, information security, and formal language theory. Many security vulnerabilities arise from discrepancies and ambiguities in interpreting input data, and such data can be considered a form of malformed language. The primary concern of LangSec is to address and correct the insecure handling and processing of input data, mitigating the risks associated with crafted-input software vulnerabilities. It is a fundamental principle to mitigate software vulnerabilities by treating the target software

[1]Translation: In his writings, a wise Italian said that the best is the enemy of the good.

DOI 10.1109/SPW63631.2024.00025

as a Turing machine for which the protocol parsers serve as the input recognition mechanism.

LangSec posits that for a system to be secure, the parsers of that system must have a well-defined, unambiguous, and finite grammar, effectively making them recognizable by machine as low as possible in the Chomsky hierarchy.

B. DFDL

DFDL (e.g., [1], [2]) is a powerful tool that enables the manipulation of structured data in a standardized and platform-independent manner. The declarative nature of its data description capabilities makes it highly valuable in various areas, encompassing data integration, transformation, validation, and legacy system integration. Consequently, it contributes to improving data interoperability and optimizing data-related operations.

DFDL encompasses additional components that expand upon the XML Schema Description Language (XSD) capabilities to provide a comprehensive means of specifying the structure and presentation of data items.

DFDL is a data modeling language that enables the precise description of data formats, regardless of their complexity. Whether we are dealing with binary data, XML, JSON, or custom data formats, DFDL allows us to define the data structure, constraints, and semantics, making it self-describing. This feature means that data consumers can understand the format and meaning of data without relying on external documentation.

Another advantage of DFDL is its platform and programming language independence. It provides a consistent way to work with data formats across different systems and technologies, making it a valuable tool for data integration, transformation, and validation tasks.

There are two versions of DFDL: an open source version released by the Apache Software Foundation [1] called Daffodil, and a commercial version sold as part of the App Connect Enterprise software by IBM® [2]. We completed our research and testing with Apache Daffodil. DFDL has the capability to handle data integration, exchange data between different platforms, convert messages, and serialize data. However, the specific aspect that is significant to us is data validation.

The DFDL framework facilitates the implementation of rigorous data validation by defining the regulations and limitations related to data formats. The use of error detection mechanisms facilitates the identification of errors and inconsistencies within the incoming data, providing informative error signals to ensure quality assurance. Therefore, we can use it to write parsers.

Furthermore, we can utilize a completed DFDL schema to generate that parser implemented in the C programming language. Subsequent source code can be compiled into a target application, resulting in a lightweight, resident native parser. The current C-code generator handles a subset of DFDL but has been sufficient for our study [6].

C. Pareto Analysis

Vilfredo Pareto (1848 – 1923) was an Italian economist and sociologist. In his 1896 book, *Cours d'économie politique* [7], Pareto observed that 20% of the population owned approximately 80% of the land in Italy. From this observation, he built a large set of income data from all over Europe and North and South America.

In the 1930s, Joseph M. Juran, a Western Electric quality engineer, noticed that the production defects were not equal in frequency. When ordering defects by frequency, he found that 20% of the defects caused 80% of the problems [8]. If the work was focused on fixing that 20%, it could have a significant impact on the defect rate with minimal effort.

The principle states that in a result, only the "vital few" cause the bulk of the results, while the more significant number of contributors, the "useful many," provide much less. Therefore, the vital few should receive priority.

One way to find the vital few contributors is to create a Pareto chart or Pareto histogram. An example is shown in Figure 1. Here, we count the number of times data elements are used and then place them in order on the graph from highest number of calls to lowest. We then draw a line (blue in the figure) showing the cumulative total or a cumulative distribution function (CDF). Looking at the line, we see that the curve's knee is at element D (red line on the figure), which divides the vital few from the useful many.

Looking at this from a coverage point of view, modification of elements A-D covers 88.7% of the total data elements, and if we narrowed that down to just elements A-C, we still cover 73.6%.

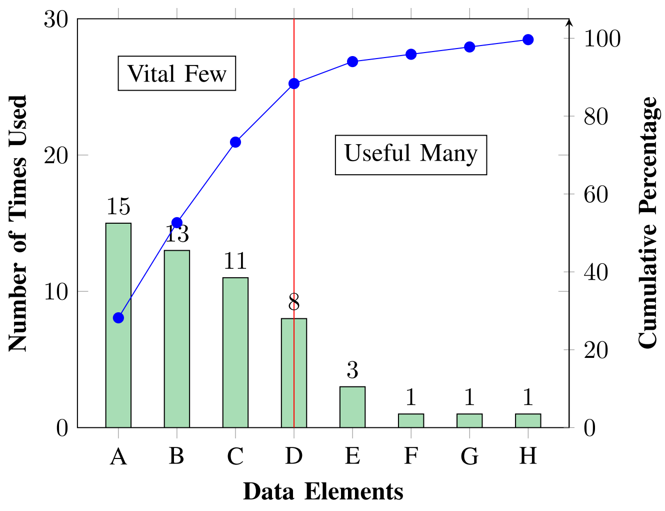


Figure 1: Sample Pareto chart, suggesting a principled way to guide allocation of limited defense resources.

D. Game Theory

Game theory (e.g., [9]) seeks to develop models to observe how various decision options affect an outcome. Originating from the work of John von Neumann and Oskar Morgenstern and the contributions of John Nash [10], this theory outlines the mathematical frameworks necessary for analyzing decision-making processes within various game structures, ranging from simultaneous and sequential games to repeated games.

The theory dissects games into fundamental components, generally players, strategies, and payoffs, and categorizes them into zero-sum, nonzero-sum, cooperative, and noncooperative games, each with distinct implications and equilibria.

Game-theoretic security (e.g., [11]) examines and analyzes the dynamics between defenders and attackers. It organizes the many ways attackers and defenders interact in strategic games. The interactions make it possible to understand conflict and cooperation in security contexts and analyze and create robust defense mechanisms.

We are interested in a simple signaling game for our research; our game has only two players: a sender and a receiver. The sender begins first by sending their signal; in our case, an NFS network packet. The receiver then observes the signal; in our case, parsing the packet and checking correctness. The two players receive or are denied a reward depending on the signal chosen by the sender and the observation of the receiver.

E. NFS

The Network File System (NFS) is a distributed remote file system protocol originally developed by Sun Microsystems in the 1980s [12]. NFS allows multiple systems to share data through a Remote Procedure Call (RPC), where programs make a call to a local library that transmits a data request on a network, which another system receives and processes, returning results or status [13]. NFS has a centralized, 1-to-n remote architecture, which supports its traditional function of providing a remote file server to a cluster of clients.

There are four major revisions of NFS, with version 4 (NFSv4) being the latest. Version 4.0, abbreviated NFSv4.0, last revised in RFC 7530 [3], is the current major release. Compared to earlier versions, it contains performance improvements, additional locking, security modes, and a stateful protocol. Version 4.1, a minor release defined in RFC 8881 [14], provides extensions for clustered servers and parallel access to files distributed over multiple servers. We studied both NFSv4.0 and 4.1.

F. Related Work

In a preliminary work-in-progress report, we demonstrated data mediation and Pareto optimization to add LangSec to software applications by evaluating the data structures of the application and their interaction with other data through its operation [15]. (This paper significantly revises and extends our earlier WIP report.)

To date, many DFDL samples tend to deal with parsing fixed record data, such as CSV, iCalendar, or GIF images [16]. Although these samples provide best-practice ways to write good DFDL code, they are of limited use for our work, as most deal with static blocks of data, where NFS requests and replies are more dynamic in size and content.

Some examples are protocols transmitted over networks, such as the National Automated Clearing House Association (NACHA) electronic payment messages, which move money between financial institutions, or ISO8583, used for card-based transactions such as point of sale. Finally, the samples also include one for Ethernet, IP, TCP, UDP, ICMP, DNS, which are raw Ethernet and the low-layer protocols that make up the backbone of Internet communication.

Strayer et al. [17] use DFDL to parse Link-16 [18], a secure communications protocol used by US and NATO.

III. METHODOLOGY

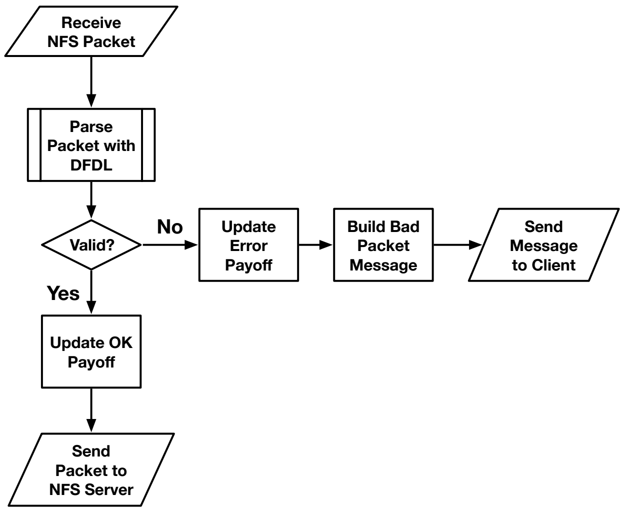
GUARDS builds on the components from Section II: we use DFDL to build LangSec filters for NFS, and then use Pareto analysis and game theory to dynamically balance mediation and performance. We call our proof-of-concept implementation *nfsfilter*, a C-based daemon designed to run on a Linux system; it is an intermediary or "bump in the wire" between the Network File System (NFS) client and the server.

The *nfsfilter* program intercepts NFSv4 commands from the client and uses Pareto analysis and game theory to assess the frequency and likelihood of error of each command. Commands identified as likely to fail are subjected to additional examination through DFDL parsing to confirm their accuracy. We transmit valid commands to the designated NFS server. Erroneous commands result in an NFS error message sent back to the client to avoid propagating possibly harmful packets to the server. By doing this, we may avoid parsing every command, which enhances performance.

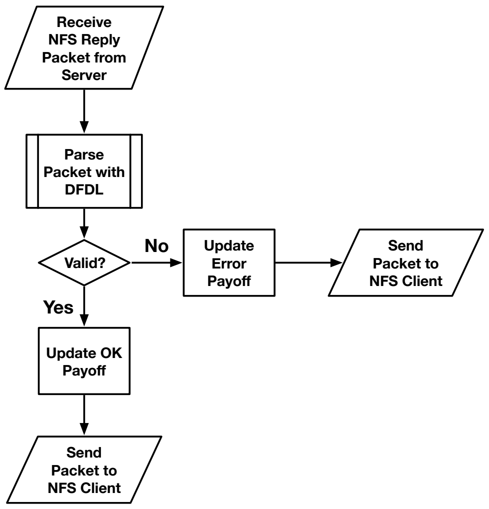
We continuously adjust our error threshold while the daemon is running. Suppose that an adversary uses different commands not currently in the error threshold to attempt to disrupt the server's operation. Since we keep a game counter for all the NFS commands sent, no matter if it is currently parsed by DFDL, increasing the number of commands may increase the counter enough to push the command above the error threshold. In addition, we use the DFDL parser to analyze the return message from the NFS server and include any faults detected by the server in the error count for each command.

The operational versatility of the *nfsfilter* daemon is an aspect of its design, allowing it to be deployed in multiple configurations to suit different network topologies and security requirements. It can function on an independent daemon on a separate system, acting as a front-end service that intermediates communication with a local NFS server. This setup enhances the security posture by isolating the NFS server from direct client interactions, providing an additional layer of scrutiny to incoming commands.

Alternatively, in a commercial setting, *nfsfilter* could be configured by an IT department to run directly on the client machine, where it interacts with the NFS server. This deployment strategy facilitates the pre-validation of commands before their transmission to the server, optimizing network traffic by filtering out erroneous or malicious requests before they physically leave the client.



(a) Client to Server


Receive NFS Reply Packet from Server
Parse Packet with DFDL
Valid?
No
Update Error Payoff
Send Packet to NFS Client
Yes
Update OK Payoff
Send Packet to NFS Client

(b) Server to Client

Figure 2: The order of operations (a) from an NFS client to *nfsfilter*, the either on to the NFS server, or return an error depending if the data parsed correctly, and (b) from an NFS server, through *nfsfilter*, and back to the NFS client. The four "update" boxes provide feedback for our dynamic reprioritization (Figure 3)

Finally, the *nfsfilter* daemon's design can be deployed directly on the NFS server system itself. In this configuration, it acts as a gatekeeper, scrutinizing and processing commands as they are received before the NFS service executes them. This deployment strategy effectively integrates the security and validation processes into the server's operational workflow, minimizing latency between daemon and server. Organizations can directly reinforce their server's security defenses by running *nfsfilter* on the NFS server, ensuring that only verified and correct commands are processed.

These three approaches enhance the security framework and leverage the daemon's flexibility to adapt to various network security architectures, safeguarding the server through an intermediary layer, and ensuring comprehensive coverage and protection across the client-server communication spectrum.

Although DFDL is a highly effective network packet parser, our approach for the daemon design is modular and could incorporate different parsers if necessary.

The remainder of this section will examine some of the ideas pertaining to the daemon's operation in greater detail.

As mentioned in Section II-D, we use a signaling game to assign points to determine whether a transaction is successful. We assign points to indicate failure rather than success; the more points an NFS operation accumulates, the more likely the command will be either frequently used or subverted.

Unlike a simple signaling game, we have no foreknowledge of whether the sender is sending good or bad input data and no direct way of discerning the sender's payoff. One way to obtain foreknowledge of the sender's packets is to design our packets specifically to get the information needed. We create special packets that work as either ideal or malicious users. Any failures are assigned additional point values.

We call "foreknowledge points" for each operation $\alpha(op)$ and $\beta(op)$. They give us the tendency of an NFS operation to be incorrect; the higher the number of points, the less reliable we believe the command is. An $\alpha(op) > 0$ means that the operation *op* was incorrectly interpreted when the data were correct. A $\beta(op) > 0$ means that the operation *op* was parsed as correct when the data were incorrect.

We run known-good and known-bad data through the daemon, then quantile bin the percentage of errors to give them values of 0 through 4. We store these as constants for the daemon to use.

Since we deal with a fixed set of NFS commands and each NFS command has a unique value, each NFS command has a game counter σ. For every command parsed correctly,

$$\sigma(op) = \sigma(op) + \alpha(op) + 1 \tag{1}$$

If the command gives an error when parsed, then

$$\sigma(op) = \sigma(op) + \beta(op) + 2 \tag{2}$$

Giving more weight to errors puts a bias into our game to prioritize errors over often-called operations. The larger the $\sigma(op)$, the more significant the potential number of errors for that command.

In Equation 1, note that we constantly add one to $\sigma(cmd)$ if correct. If we have an implied perfect system, that is, $\alpha(cmd) = \beta(cmd) = 0$, we will rely on the commands used the most to be the ones that are Pareto selected until the system collects unexpected errors.

If $\alpha(cmd) \neq 0$ or $\beta(cmd) \neq 0$, we should closely monitor the command cmd as it has demonstrated a tendency to have an error. Its $\sigma(cmd)$ will grow faster, which means that it has a higher chance of being above τ and constantly being tested by LangSec.

Suppose that a client's command packet fails the NFS parser. In that case, the *nfsfilter* module will generate a "bad data" reply packet and transmit it back to the client, prohibiting potentially malicious data transmission to the NFS server. If the command packet is correct, *nfsfilter* transmits it to the NFS server for further processing. The point value for that command updates depending on the outcome.

Figure 2a illustrates the transmission of packets by an NFSv4 client on port 2049, the NFSv4 interface port. The filter service operates on port 2049 to conceal the NFS server's true identity from the clients.

In Figure 2b, the NFS server returns the status to *nfsfilter*, which parses the status for correctness before sending it back to the client. The point value for the command's reply packet updates depending on the outcome.

τ is our point value threshold, where operations with a $\sigma(op) > \tau$ will be fully parsed and those below τ will not be. The initial setting for τ is 0, allowing the DFDL parser to process every command; over time, τ will increase as *nfsfilter* runs, first by the commands called the most, then the commands containing errors will start to bubble up to the top.

We recalculate τ when the program is in an idle state and the last recalculation was more than 15 seconds ago. Figure 3 shows a flow chart of this operation. We calculate a Pareto CDF by ordering the current points of all the operations that are greater than zero, and then calculating the knee of the curve. If the difference between the old and new values exceeds the tolerance, the new value becomes τ.

One way to find the knee of a Pareto CDF is to calculate the perpendicular distance from each point on the curve to the line representing the linear decrease from the first to the last point on the curve; we use the L-Method [19] to calculate this. The point on the curve with the maximum perpendicular distance to the line is the knee of the curve.

Finally, we implemented a SQLite database to store the data for each operation and the current τ. SQLite reads the information from permanent storage at initialization and stores it in a structure in the application. The updated information is returned to the storage medium before the application terminates.

Limitations

Our approach currently has some limitations. While DFDL is robust, the DFDL to C converter does not currently handle all the commands and attributes available in the language. Due to these limitations, we cannot fully realize some NFS operations in our model and must make some adjustments. For example, the *dirlist4* structure used by the *readdir* command is a linked list structure that does not supply a count, only an end variable equal to zero for the last entry. While DFDL can set an unlimited count for the structure, the C converter cannot. We generated comments with any limitations to our parser's DFDL code.

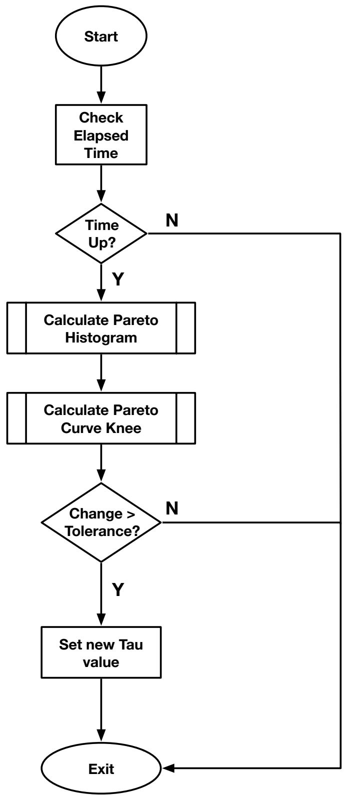


Figure 3: Calculating a new τ in *nfsfilter*, using the feedback updates from Figure 2.

The accurate representation of every NFS operation and the detection of out-of-bounds values in that operation are crucial for detecting any modifications that an attacker introduces. Misses in detection lie not with DFDL itself but rather with the model developer's responsibility to ensure strict adherence to the NFS standards.

We designed *nfsfilter* as a demonstration application; therefore, it is not ready to use on production systems or networks.

IV. Results

For our tests, we set up *nfsfilter* as a daemon on the NFS client that interfaces with the NFS server. Although the daemon can accept more than one connection, we run only one client connection to collect accurate timing.

Our NFS client is a Dell XPS 13 laptop with an i7-7500U, 2.7GHz processor, and 16GB of RAM. Our NFS server is an HP Zbook 17 laptop with a Core i7-4910MQ, 2.9GHz processor, and 32GB of RAM. Both test systems are Linux-based and run Ubuntu 22.04, with all the latest updates.

Both systems connect to a 1Gb Ubiquiti USW-Lite-16-PoE network switch; we did not isolate the switch from the backbone network, but the switch management system showed off-switch interactions averaging 254Kbps during testing. The *nfsfilter* daemon receives connections on the client's localhost (127.0.0.1) at port 2049 and forwards accepted packets to and from the server's NFS port 2049.

Our toolchain uses Apache Daffodil version 3.7.0-SNAPSHOT for DFDL and NFS versions 4.0 and 4.1 for NFS connections. The *clang* C compiler version 15.0.7 with *-O3* optimization was used to compile *nfsfilter*.

A. Scenario Timing Tests

To test the latency of adding *nfsfilter*, we recorded NFS sessions between a local client and a server (direct) and a local client running through *nfsfilter* (indirect). Each scenario is a bash shell script that uses standard Linux commands that interact with the NFS server. The scenario's time was collected by running Wireshark, filtering for NFS packets on the appropriate interface, and collecting the clock times. We run each scenario with τ set to zero, (i.e., all packets checked) five times, and then we took the mean of the results. We performed the following scenarios:

- Mount a directory, get its listing, and unmount.
- Mount a directory, read a 70KB file, and unmount.
- Mount a directory, remove a file, write a new 70KB file, and unmount.

We use both NFS versions 4.0 and 4.1 to test the scenarios, as they use different underlying commands to mount and unmount directories. By testing both versions, we test the broadest command set possible.

The results and the reduction in throughput using *nfsfilter* are in Table I. At the bottom of the table, we show the mean for all tests, where our aggregate time to parse 100% of the NFS operations ranged from 54.7% to 113.9% slower than connecting directly to the NFS server. We can attribute some of the speed issues to sending and receiving through the *nfsfilter* daemon and some inefficiencies in coding the daemon when reading and writing larger files.

The results did not give us the best view into how well our mediation system worked, so we changed the daemon's session handler to capture the session's time. We read the processor time with *clock()* when a client session begins after connecting to the server but before the handler processes the first command. We reread the processor time when the client closes the session and calculate the difference; these calculations show only the work done by the parser code without any network overhead.

We used the three scenarios listed above, setting τ to zero, running each scenario five times, and then taking the mean of the results. We modified the session handler to not parse any commands, parse all commands, and parse only 50% of the commands to see what the parser costs us in speed.

Table II shows the time spent in the session handler sending and receiving commands. Examining the mean at the bottom of the table, we can see that allowing 100% mediation costs approximately 44% more than simply passing the data through without parsing. If we only parse 50% of the data, the cost drops by half in the worst case and by nearly six times in the best case.

The variation in parsing speed is due to the differing complexities of each NFS operation; some have more variables to parse than others. Combining the data from both tables, we observe that our parsing cost is only 7.3% of the mean for NFSv4.0 and 4.7% for NFSv4.1 based on 100% parsing. Note that by mediating our parsing to only the most used or most error-prone operations, the percentage of operations parsed will probably be less than 50%; the number of operations is fixed (37 in NFSv4.0 and 51 in NFSv4.1), and many are specialized operations and not used in day-to-day operation. In our test network of two systems, we use 10 to 12 operations the most, so we might only mediate 25% to 33% of the commands at worst. We discuss this more in mediation testing.

B. Mediation Testing

Mediation testing assesses the ability of our algorithm to identify NFS operations that are prone to errors or have a high likelihood of errors. We created a scenario that mounts the remote directory, reads a file in the initial directory, navigates to a different directory, deletes a file, writes a new version, and finally unmounts the directory. We chose this scenario to exercise the commonly used NFS operations. We will utilize our read/write scenario with versions 4.0 and 4.1 to evaluate the different NFS operations used in session management.

We ran our test scenario from a script, then captured the packets using Wireshark so we could play them back with *tcpreplay*. Binary NFS packets allowed us to perform some simple data fuzzing; we can easily insert errors and send those packets to *nfsfilter* for parsing.

Our first test did not add foreknowledge points to the command base; every operation parsed correctly, so we only collected usage counts. We ran our test scenario with NFSv4.1; Table III shows the commands called. Calculating τ from the Pareto CDF reveals the knee of the curve at GETATTR, shown in Figure 4, so only the SEQUENCE, PUTFH, and GETATTR commands parse, and the other commands pass through *nfsfilter*. Our parsing cost is approximately 5.9% as we mediate 3 out of the 51 possible operations NFSv4.1 has.

We then reran this test but added ten CLOSE operations to the end of the sequence. Since the test sequence had already closed the files and terminated the session, the NFS server returned the error "no filehandle" as part of its reply to the client. *nfsfilter* parses the reply, reads the error, and updates the error count for CLOSE. With no foreknowledge set, we collect 2 points for each error, so CLOSE now has 21 points, one correct CLOSE plus 10 in error. CLOSE is now above the current τ threshold, so all CLOSE commands from this point are fully parsed. When we next recalculate τ, these data are taken into account, as shown in Figure 5. Our parsing cost increased to 7.8%; we now mediate 4 out of the 51 possible operations.

V. Discussion and Future Work

As seen in the prior section, our parser is efficient, but we need to enhance the efficiency of the remaining parts of the daemon. *nfsfilter* needs to test its alternative modes of operation: one on a dedicated server and the other as a front-end to the NFS server; this may lead to increased efficiency compared to running it on the client. We will refactor or

Test	No Daemon		With *nfsfilter*, 100% Parsing (Percent slower)	
	v4.0	v4.1	v4.0	v4.1
Directory list	35.4 ms	80.2 ms	60.0 ms (69.5%)	124.4 ms (55.1%)
File read	48.2 ms	97.2 ms	137.2 ms (184.6%)	163.6 ms (68.3 %)
Delete, then write a file	50.0 ms	88.6 ms	88.6 ms (77.2%)	123.6 ms (39.5%)
Mean of tests	44.5 ms	88.7 ms	95.3 ms (113.9%)	137.2 ms (54.7%)

Table I: Scenario Timing Tests for complete mediation where the time is calculated by obtaining start and stop times from Wireshark, so this describes the whole round-trip on the network.

Test	v4.0			v4.1		
	No Parse	Parse 100%	Parse 50%	No Parse	Parse 100%	Parse 50%
Directory list	4.42 ms	6.3 (42.5%)	4.68 (5.9%)	3.56 ms	5.86 (64.6%)	4.78 (34.3%)
File read	4.34 ms	5.48 (26.3%)	4.34 (0%)	4.04 ms	5.72 (41.6%)	4.34 (7.4%)
Delete, then write a file	5.6 ms	8.98 (60.4%)	6.42 (14.6%)	5.48 ms	7.26 (32.5%)	6.9 (25.9%)
Mean of tests	4.79 ms	6.92 ms (44.6%)	5.15 ms (7.5%)	4.36 ms	6.28 ms (44.0%)	5.34 ms (22.5%)

Table II: Using the Scenario Timing Tests to calculate latency in the parser for *nfsfilter*. We display the times for no parsing, parsing all operations, and parsing half the operations. The numbers in parentheses are the percent slower than no parsing.

Operation	Usage Count
SEQUENCE	16
PUTFH	13
GETATTR	12
ACCESS	2
EXCHANGE_ID	2
GETFH	2
PUTROOTFH	2
CLOSE	1
CREATE_SESSION	1
DELEGRETURN	1
DESTROY_CLIENTID	1
DESTROY_SESSION	1
NULL	1
OPEN	1
READ	1
RECLAIM_COMPLETE	1
SECINFO_NO_NAME	1

Table III: Initial NFS4.1 mediation test with τ and foreknowledge points equaling zero.

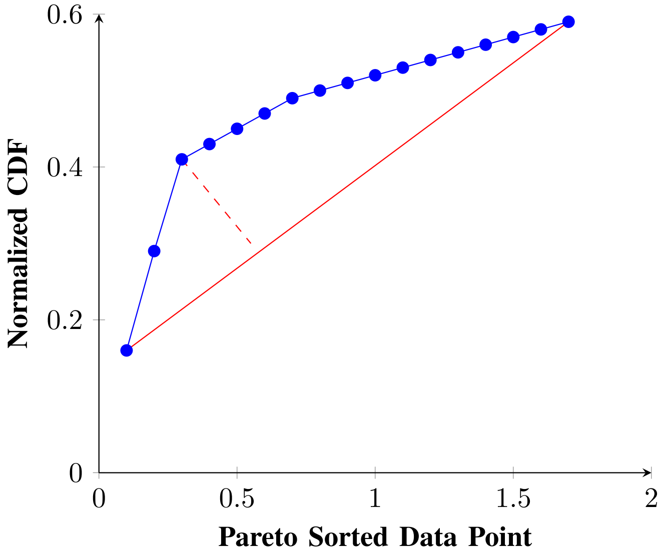


Figure 4: Pareto CDF finding the knee of the initial mediation test.

rework certain *nfsfilter* networking functions to enhance their efficiency, mainly to move beyond a proof-of-concept.

More involved mediation tests will also occur. We plan to take the current scenarios and run them through a dedicated fuzzer such as AFLNET [20], which will mutate the scenario packets and present them to *nfsfilter* for processing.

We must also assess our effectiveness in detecting adversaries attempting to compromise or crash the daemon or send malicious packets to the NFS server through out-of-band or other novel methods.

For example, one issue of concern is slow delivery rate attacks, where an attacker delivers carefully constructed packets at random intervals to disrupt or confuse a server. Another area is slow or low-rate Denial of Service (DoS) attacks, such as SlowDrop [21], which emulate multiple nodes interacting with a server through an inconsistent network connection rather than overwhelming it with a large volume of packets at once like the usual DoS attack.

Further research is needed to evaluate security against an adversary who is knowledgeable about the functioning of dynamic mediation; one objective is to classify the harmful data transmitted by an attacker. A practical approach involves using a pseudo-hash, such as Simhash [22], as a unique identifier to recognize similarities in incorrect commands transmitted promptly. Although Simhash is best known for fast searches for similar documents in large-scale collections, a version of Simhash is also employed to detect malicious data in network traffic monitoring and security evaluation (e.g., [23], [24]).

Finally, it would be beneficial to incorporate external data, such as vulnerability announcements, to update the foreknowledge points and adjust the prioritization of the mediation

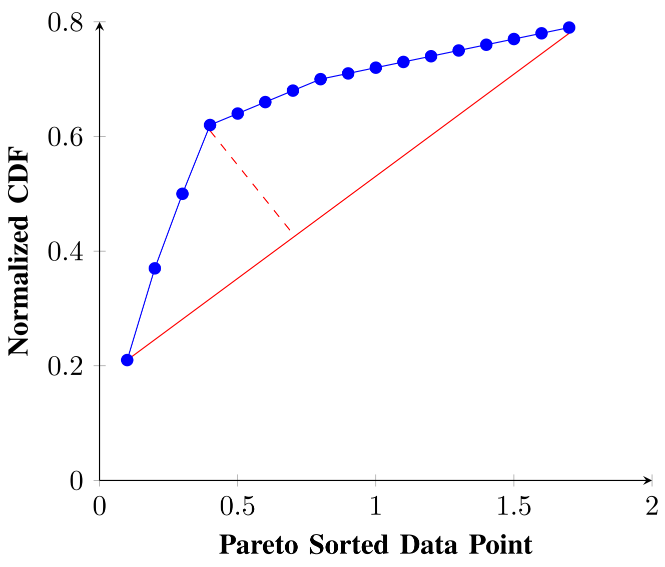


Figure 5: Pareto CDF finding the knee of the mediation test with CLOSE errors.

process for one or more NFS operations.

VI. Conclusion

Our initial work showed that DFDL is a powerful tool to define and model the structure of various data formats. We have shown that these formats include network parsers that can precisely analyze data streams such as NFS. This approach allows for the precise definition of NFS procedures, minimizing the risk of errors that may jeopardize the data's integrity or the system's security. DFDL's versatility in defining data formats makes it crucial for designing efficient network parsers that are both effective in processing data and adaptable to changes in network protocols or data structures.

Improving our basic DFDL parser, GUARDS introduced a dynamic system designed to prioritize the NFS procedures most likely to encounter errors, providing a principled way to improve the performance of the system while still providing protection. By optimizing procedures with the highest probability of errors, GUARDS can preemptively address issues before they result in failures or security breaches. This proactive approach improved the overall stability and security of the NFS environment.

Acknowledgment

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001119C0121. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA).

The authors thank John Interrante at GE Research for his help providing best practices for creating DFDL code which supported our continued development. We also thank William Smith and the GE GAPS team for their support.

The authors also thank the paper's reviewers for taking the time to improve our work.

The source code for GUARDS and any subsequent changes can be accessed at https://github.com/jpbdart/guards.

References

[1] Apache Software Foundation, "Apache Daffodil," 2023. [Online]. Available: https://daffodil.apache.org

[2] IBM, "IBM App Connect Enterprise, Version 11.0.0.16 – DFDL," 2022. [Online]. Available: https://www.ibm.com/docs/en/app-connect/11.0.0?topic=model-data-format-description-language-dfdl

[3] T. Haynes and D. Noveck, "Network File System (NFS) Version 4 Protocol," RFC Editor, RFC 7530, Mar. 2015. [Online]. Available: https://tools.ietf.org/html/rfc7530

[4] Voltaire, "La Bégueule," in *Romans de Voltaire suivis de ses contes en vers*. Garnier, 1922, pp. 541–548.

[5] F. Momot, S. Bratus, S. M. Hallberg, and M. L. Patterson, "The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them," in *2016 IEEE Cybersecurity Development (SecDev)*, Nov. 2016, pp. 45–52.

[6] M. Beckerle and J. Interrante, "Daffodil Code Generators - DAFFODIL - Apache Software Foundation," Feb. 2023. [Online]. Available: https://cwiki.apache.org/confluence/display/DAFFODIL/Daffodil+Code+Generators

[7] G. Bousquet and G. Busino, *Cours d'Économie Politique: Nouvelle édition par G.-H. Bousquet et G. Busino*. Librairie Droz, 1964, vol. 1.

[8] J. A. D. Feo, *Juran's Quality Handbook: The Complete Guide to Performance Excellence, Seventh Edition*. McGraw-Hill Education, 2017. [Online]. Available: https://www-accessengineeringlibrary-com.dartmouth.idm.oclc.org/content/book/9781259643613

[9] M. J. Osborne, *An Introduction to Game Theory*. New York: Oxford University Press, 2000, vol. 3.

[10] R. J. Leonard, "From Parlor Games to Social Science: Von Neumann, Morgenstern, and the Creation of Game Theory 1928-1944," *Journal of Economic Literature*, vol. 33, no. 2, pp. 730–761, 1995, publisher: American Economic Association. [Online]. Available: https://www.jstor.org/stable/2729025

[11] M. Tambe, *Security and Game Theory*. Cambridge University Press, 2011.

[12] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation of the Sun network filesystem," in *Proceedings of the Summer USENIX conference*, 1985, pp. 119–130.

[13] A. D. Birrell and B. J. Nelson, "Implementing Remote procedure calls," in *Proceedings of the ninth ACM symposium on Operating systems principles*, ser. SOSP '83. Association for Computing Machinery, Oct. 1983, p. 3. [Online]. Available: https://doi.org/10.1145/800217.806609

[14] D. Noveck and C. Lever, "Network File System (NFS) Version 4 Minor Version 1 Protocol," RFC Editor, RFC 8881, Aug. 2020. [Online]. Available: https://tools.ietf.org/html/rfc8881

[15] J. P. Brady and S. W. Smith, "Work In Progress: Optimizing Data Mediation Placement with Pareto Analysis," in *The Seventh IEEE Workshop on Language-Theoretic Security (LangSec 2021)*, May 2021. [Online]. Available: https://github.com/gangtan/LangSec-papers-and-slides/raw/main/langsec21/papers/Brady_LangSec21.pdf

[16] smh@uk.ibm.com, "DFDL Schemas for Commercial and Scientific Data Formats." [Online]. Available: https://github.com/DFDLSchemas

[17] T. Strayer, R. Ramanathan, D. Coffin, S. Nelson, M. Atighetchi, A. Adler, S. Blais, B. Thapa, W. Tetteh, V. Shurbanov, K. Haigh, R. Hain, C. Rock, E. Do, A. Caro, D. Ellard, M. Beckerle, S. Lawrence, and S. Loos, "Mission-Centric Content Sharing Across Heterogeneous Networks," in *2019 International Conference on Computing, Networking and Communications (ICNC)*, Feb. 2019, pp. 1034–1038, iSSN: 2325-2626. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8685557

[18] R. Riley, "Providing link-16 on disadvantaged platforms using the multi-role advanced tranceiver | IEEE Conference Publication | IEEE Xplore," Oct. 2007. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4391913

[19] S. Salvador and P. Chan, "Determining the number of clusters/segments in hierarchical clustering/segmentation algorithms," in *16th IEEE International Conference on Tools with Artificial Intelligence*. Boca Raton, FL, USA: IEEE Comput. Soc, 2004, pp. 576–584. [Online]. Available: http://ieeexplore.ieee.org/document/1374239/

[20] V.-T. Pham, M. Böhme, and A. Roychoudhury, “AFLNET: A Greybox Fuzzer for Network Protocols,” in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, Oct. 2020, pp. 460–465, iSSN: 2159-4848. [Online]. Available: https://ieeexplore.ieee.org/document/9159093

[21] E. Cambiaso, G. Chiola, and M. Aiello, “Introducing the SlowDrop Attack,” *Computer Networks*, vol. 150, pp. 234–249, Feb. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128619300210

[22] C. Sadowski and G. Levin, “SimHash: Hash-based Similarity Detection,” UC Santa Cruz, Tech. Rep. UCSC-SOE-11-07, Feb. 2011. [Online]. Available: https://tr.soe.ucsc.edu/sites/default/files/technical-reports/UCSC-SOE-11-07.pdf

[23] R.-H. Dong, C. Shu, and Q.-Y. Zhang, “Security Situation Assessment Algorithm for Industrial Control Network Nodes Based on Improved Text SimHash,” *International Journal of Network Security*, vol. 23, no. 6, pp. 973–984, Nov. 2021. [Online]. Available: https://www.airitilibrary.com/Article/Detail/18163548-202111-202111020002-202111020002-973-984

[24] Y. Li, F. Liu, Z. Du, and D. Zhang, “A Simhash-Based Integrative Features Extraction Algorithm for Malware Detection,” *Algorithms*, vol. 11, no. 8, p. 124, Aug. 2018, number: 8 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/1999-4893/11/8/124