Bachelor's Thesis Computing Science

Radboud University Nijmegen

Differential fuzzing of stateful systems using FlexFringe

Author:
Jermo van Oort
s1058152

First supervisor/assessor:
Dr. Ir. Erik Poll

Second assessor:
Prof. Dr. Frits Vaandrager

Second supervisor:
Cristian Daniele

Third supervisor:
Seyed Behnam Andarzian

January 19, 2024

Abstract

In this day and age, all users of the Internet use stateful protocols to transfer messages with one another. Stateful protocols keep track of the communication's current state, often employed to maintain a connection between two devices. To keep users safe, all server implementations of these protocols need to be as secure as possible to prevent adversaries from misusing them. An effective method for discovering vulnerabilities in a protocol's server implementation is fuzzing. Fuzzers operate by feeding an implementation large amounts of randomly generated data, looking for unexpected crashes or outputs. A state-of-the-art fuzzer is AFLNet, designed specifically for stateful protocols.

Generating state machines from the outputs of the fuzzer and subsequently comparing them results in differential fuzzing. Differential fuzzing offers a potential solution for finding differences, and hence bugs, in implementations of the same protocol. By feeding the inputs and outputs that were generated by the fuzzer to a state machine learner as FlexFringe, different implementations of a stateful protocol can be compared based on their state machines. Differences can indicate vulnerabilities or newly introduced bugs.

Using AFLNet to fuzz two different server implementations of FTP and then pass the traces to FlexFringe to generate the state machines revealed that there are noticeable differences between implementations, both in the number of states and the ways they deal with authentication. Furthermore, inspecting the state machines for vulnerabilities uncovered certain inputs that have the potential to be security risks.

Contents

Chapter 1

Introduction

Differential testing is valuable as it can expose differences and flaws among various software implementations by comparing their behavior. It provides confidence in the reliability and consistency of applications, by uncovering bugs and thus potential security vulnerabilities [16]. Another method for discovering bugs is with the use of a fuzzer. Fuzzing is done by feeding a system under test (SUT) large amounts of randomly generated data, looking for unexpected crashes or outputs [12]. Integrating fuzzing with differential testing leads to differential fuzzing. Differential fuzzing is achieved by fuzzing a SUT and comparing various software implementations with each other. This technique will be implemented on stateful protocols in this study. Stateful protocols, such as FTP and TCP, have a wide variety of server implementations and versions. When these protocols are newly implemented, they are essentially built from the ground up, potentially resulting in minor variations. These discrepancies can become vulnerabilities that adversaries can exploit to carry out attacks. The variations can be compared with a finite state machine learner like FlexFringe [27].

By choosing a fuzzer that works with stateful protocols [11], the input and output traces are used as input for FlexFringe to create a state machine. Traces are the client input and server output of an implementation. The divergences between state machines of different implementations can indicate bugs or inconsistencies. The following research question will be answered at the end of this study: **How to perform differential fuzzing of a stateful protocol using FlexFringe?**

Chapter 2 will provide background information to understand some technical details used in this thesis. This includes information about fuzzing in general and different learning algorithms for state machine learning. Chapter 3 discusses the choice of fuzzer. Chapter 4 presents the choice of the used stateful protocol and their implementations. Chapter 5 consists of two phases. In the first phase, fuzzing LightFTP and Bftpd with AFLNet [19] is done. In the second phase, the process of finding the right abstraction

function and using the correct heuristics is given. Chapter 6 contains the final state machines for the two FTP server implementations and both implementations are discussed. Chapter 7 lists the work that can be done in the future to extend this research. Lastly, Chapter 8 presents the conclusions of the research findings. Figure 1.1 gives an abstract flow of how the state machines are created.

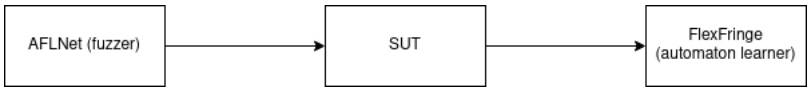


Figure 1.1: Abstract flow of this research

Chapter 2

Preliminaries

In this chapter, three different approaches for fuzzing are explained in section 2.1. This section also includes the definitions of differential testing and fuzzing. In section 2.2 active and passive learning algorithms to learn state machines are discussed.

2.1 Fuzzing

Fuzzing is all about sending various inputs to the system under test (SUT) and observing if bugs or incorrect outputs occur. There are multiple approaches to fuzzing. In sections 2.1.1, 2.1.2, and 2.1.3 three different approaches are explained. Lastly, differential testing and fuzzing are explained in more detail in section 2.1.4.

2.1.1 Blackbox fuzzing

In blackbox fuzzing, the SUT is known as a box where the source code is not known [29] and the fuzzer must solely rely on the SUT's external behavior and responses. The input that must be provided to the SUT does not have any restrictions other than being a finite string, but in practice, the input is crafted in such a way that it finds bugs more easily, like using grammar- or mutation-based fuzzers. Grammar-based fuzzers use a predefined syntax or grammar that meets the expected input structure [8]. By generating according to the grammar, it is more likely that the inputs trigger specific code within the SUT. Mutation-based fuzzing makes small and random alterations to already existing inputs to generate new inputs. The output of the SUT will be checked for any divergences that could indicate a bug and thus might be a security flaw.

2.1.2 Whitebox fuzzing

Whitebox fuzzing is exactly the opposite of blackbox fuzzing. With whitebox fuzzing, the fuzzer can access the source code of the SUT and the fuzzer can use the internal behavior to create input. To maximize the efficiency of the fuzzer, inputs will be crafted so that they reach all branches in the code. This is often done using symbolic execution [10]. Symbolic execution is a program analysis technique used to find the behavior of a program by following the values of variables and expressions instead of random input. It is valuable in identifying edge cases and can automatically generate input for said edge cases.

2.1.3 Greybox fuzzing

Greybox fuzzing is in between white- and blackbox fuzzing. Greybox fuzzers make use of observations made during the execution stages of fuzzing to guide the fuzzer to the highest possible coverage tree. Trying to achieve this coverage is not done by simply generating data and hoping for the best. Instead, the fuzzer learns from the SUT's responses and uses this knowledge to guide the process. Focusing on prioritizing inputs that expand the coverage of the SUT and refine these through mutation [6].

Greybox fuzzers, such as AFLNet [19], further enhance their efficiency by using two feedback approaches to steer the fuzzing process. The first is a coverage-driven approach and a second is a response-driven approach.

The coverage-driven approach is the most common form of greybox fuzzing. Take for example code-coverage feedback. Code-coverage feedback emphasizes the selection and prioritization of inputs that lead to the execution of undiscovered code [25]. Other common coverage metrics are line, function, edge, or path coverage.

The response-driven approach on the other hand uses the response of the system to steer the process. The inputs that result in previously undiscovered responses can be used and modified to broaden the fuzzer's scope even further [19].

A state-of-the-art greybox fuzzer for stateful protocols is AFLNet [19], which is extensively discussed in section 3.1

2.1.4 Differential testing and fuzzing

Differential testing is a technique used to compare the behavior of two or more implementations of the same software or protocol, aiming to identify differences in how these implementations handle data and different inputs [5]. The testing process consists of sending the same input, including invalid and edge case inputs, to both implementations and comparing the behavior of the program. Do both implementations produce the same output?

Different outputs can indicate a bug or non-standard behavior. Differential testing is valuable in implementation-specific flaws or inconsistencies. Comparing the two implementations is different for stateless and stateful protocols. For stateless protocols, only the input and output can be compared with one another, because the SUT is a box with no states in it. All these inputs and outputs are stored in traces. These traces are compared with the traces from the other implementation. For stateful protocols, there are internal states. Comparing two different implementations can be done by comparing the state machine of the implementations. Differences in state machines, e.g. a transition that is in one that is missing in the other, can indicate undesired behavior. Differential fuzzing is an extension of differential testing because it automatically tests the SUT with loads of different random inputs generated by a fuzzer. The inputs are sent to the SUT, and the outputs are monitored.

2.2 Learning algorithms for state machines

In section 2.2.1 it is explained what active learning is. Section 2.1.2 gives the details about the passive learning approach.

2.2.1 Active learning

During the learning process, where the learner attempts to construct a state machine from the obtained traces, active learning algorithms can interact with the SUT [22]. When learning state machines, the protocol can send necessary input traces to the SUT and instantly receive the corresponding outputs. This approach can facilitate the learning of state machines while interacting with the SUT.

2.2.2 Passive learning

Passive learning does not rely on actively selecting traces to send to the SUT. Instead, it uses collected traces to construct a state machine [22]. This approach is useful when dealing with a large quantity of data because passive learning is less time-consuming than active learning [22]. Active learning algorithms need to label the input for the next iteration as the previous iteration finishes, passive learning can do that beforehand [14]. In section 5.2.1, FlexFringe [27], a passive state machine learner is discussed.

Chapter 3

Choice of fuzzer

In this chapter, the choice of the fuzzer will be discussed. Firstly, AFLNet [19] is discussed. AFLNet, built upon Google's American Fuzzy Lop (AFL)[1], is a greybox fuzzer specifically designed for stateful protocols. Details about AFLNet are provided in section 3.1. Another stateful protocol fuzzer, BooFuzz, is discussed in section 3.2. In section 3.3 the choice for AFLNet is explained.

3.1 AFLNet

AFLNet[19] is a greybox fuzzer based on the AFL groundwork, inheriting and expanding upon the properties implemented by AFL. One of the characteristics of AFLNet is that it has the capability to fuzz stateful programs, in contrast to AFL, which is more optimized for stateless programs. Furthermore, AFLNet takes a dual feedback approach, using both response feedback and code-coverage feedback (section 2.1.3) to steer the fuzzing process. The fuzzer acts as a client that sends messages to the server (SUT) and modifies and replays messages that are effective at increasing the coverage or state space. AFLNet needs an initial set of recorded messages to the system under test (SUT), upon which it builds and progresses. AFLNet is able to fuzz in parallel, which increases the efficiency of the fuzzing process [7].

3.2 BooFuzz

BooFuzz [18] is a grammar-based blackbox fuzzer that is a fork of the now inactive Sulley fuzzer [3]. BooFuzz can fuzz stateful protocols, and combine that with quick data generation [18], making it well-suited for protocols with complex state machines. BooFuzz has loads of documentation[2] about

[1] https://github.com/google/AFL
[2] https://boofuzz.readthedocs.io/en/stable/

the installation process and how the software must be used. Unfortunately, BooFuzz is a blackbox fuzzer, so it cannot make use of the source code or observations made in the execution phase. To fuzz a stateful protocol that uses any sort of authentication, some variables need to be known before fuzzing. Before fuzzing with BooFuzz, a session is created that includes a username and password. Boofuzz can fuzz multiple targets in parallel [18], but cannot fuzz the same job on multiple cores or machines.

3.3 Choice of fuzzer

In this study, AFLNet will be used as fuzzer. AFLNet is a greybox fuzzer instead of the blackbox fuzzer BooFuzz. Therefore, it uses the observations made in the execution phase of the process, which leads to the execution of undiscovered code faster (section 2.1.3). Adding the use of seed input gives the fuzzer an understanding of the syntax for that protocol. Furthermore, AFLNet can fuzz jobs on multiple cores or machines, hence decreasing the time before all states are found. AFLNet is used in this study.

Chapter 4

Choice of protocol and implementations

In the chapter, different protocols are discussed. All protocols have different characteristics. In section 4.1 the characteristics of FTP are discussed, and different FTP server implementations are talked about. In section 4.2 TCP will be discussed. Lastly, in section 4.3 the choice for the protocol is explained.

4.1 FTP

File Transfer Protocol (RFC 959[2]), designed in 1971, is a stateful protocol specifically made to transfer files across the internet. In the section 4.1.1, LightFTP is discussed. LightFTP is a lightweight FTP server that can communicate with FTP clients. Then in section 4.1.2, ProFTPD is talked about. Lastly, in section 4.1.3, Bftpd is discussed.

4.1.1 LightFTP

LightFTP[9] is a lightweight FTP server suitable for small-scale file transfer across the internet. An FTP client can connect to the FTP server and put or get files. In listing 4.1 an example FTP message exchange is given.

```
1 220 LightFTP server ready
2 USER anonymous
3 331 User anonymous OK. Password required
4 PASS password
5 230 User logged in, proceed.
6 PWD
7 257 "/" is a current directory.
8 QUIT
```

```
221 Goodbye!
```

Listing 4.1: Message exchange of FTP client and LightFTP server. Odd lines are server responses (output traces), and even lines are client commands (input traces).

This example lets the client log into the server with a username and password and when the client is authenticated, the client requests the path to this directory. Lastly, the client quit from the server. The messages with FTP commands (like USER and PASS) are commands from the client. The other messages are responses from the server, indicated with a code.

FTP has a fixed list of FTP commands.

4.1.2 ProFTPD

ProFTPD [21] is an open-source FTP server with as goal to have as many features and configurations as possible so that users have options to choose from. ProFTPD has modules that allow for encryption for file transfers. ProFTPD uses standard FTP input and response codes for communicating with a client.

```
220 ProFTPD Server (ProFTPD Default Installation) [127.0.0.1]
USER ubuntu
331 Password required for ubuntu
PASS ubuntu
230 User ubuntu logged in
DELE test.txt
550 test.txt: No such file or directory
QUIT
221 Goodbye.
```

Listing 4.2: Message exchange of FTP client and ProFTPD server. Odd lines are server responses (output traces), and even lines are client commands (input traces).

This example lets the client log into the server with a username and password. The client tries to delete a file, but that file does not exist.

4.1.3 Bftpd

Another small and easy-to-configure FTP server is Bftpd [24]. Bftpd's aim is to be fast, secure, and quick to set up and configure FTP servers. Some key features are: no special setup is needed for security with chroot and files (sh, ls, ...) are not needed in the chroot environment. Most FTP commands are implemented in Bftpd.

```
220 bftpd 5.7 at 127.0.0.1 ready.
USER ubuntu
331 Password required for ubuntu
PASS ubuntu
```

```
230 User ubuntu logged in
SYST
215 UNIX Type: L8
PWD
257 "/" is a current directory.
QUIT
221 See you later...
```

Listing 4.3: Message exchange of FTP client and Bftpd server. Odd lines are server responses (output traces), and even lines are client commands (input traces).

4.2 TCP

Another stateful protocol is the transmission control protocol (RFC 793 [1]). TCP, designed in 1981, is a reliable host-to-host protocol. Detection mechanisms are in place for identifying out-of-sync packets or communication errors. TCP is widely employed, with even other protocols relying on the connections established by TCP. Due to its extensive usage, there are plenty of diverse implementations of this protocol available in nearly every programming language.

4.3 Choice of protocol

For this thesis, only the FTP protocol is used. There are a lot of different easy-to-setup and configure FTP server implementations, so that is why FTP is chosen. TCP is an alternative to look at in the future (section 7.2). In Chapters 5 and 6, only (the creation of) the state machines of LightFTP and Bftpd are given. ProFTPD is not handled due to the fuzzing results, where the number of input traces exceeded the output traces in almost all instances. This made the creation of the abstraction function very difficult (section 5.2.5).

Chapter 5

Setting AFLNet and FlexFringe for FTP

This chapter outlines the process of fuzzing LightFTP and Bftpd using AFLNet, as well as experimenting with the creation of different state machines. Figure 5.1 schematically shows how the state machines are constructed. The process indicated by Phase 1 is discussed in section 5.1. This section is about fuzzing the different implementations and collecting the traces. In section 5.2, Phase 2 is described as indicated in Figure 5.1. Experiments with different heuristics and abstraction functions are done to find the right configurations for generating state machines.

5.1 Fuzzing FTP implementations with AFLNet

In this section, it is explained how to install and fuzz different FTP implementations with AFLNet. Firstly, setting up AFLNet by running the Dockerfile is demonstrated in section 5.1.1, then the prerequisites that AFLNet needs to start fuzzing any FTP implementation are given in section 5.1.2. After that, in section 5.1.3, errors that could occur during the fuzzing process are tackled. Subsequently, in section 5.1.4 LightFTP is fuzzed, followed by Bftpd in section 5.1.5. Lastly, it describes how to capture traces in section 5.1.6.

5.1.1 Setting up AFLNet

AFLNet GitHub repository[1] has excellent documentation on how to install AFLNet. It is advisable to install AFLNet using the provided Dockerfile in the repository or via Appendix A, as attempting to install it directly on Ubuntu 22.04 has resulted in numerous errors and may not work at all. In my experience, the Dockerfile simplifies the setup and ensures a reliable and easy

[1] https://github.com/aflnet/aflnet

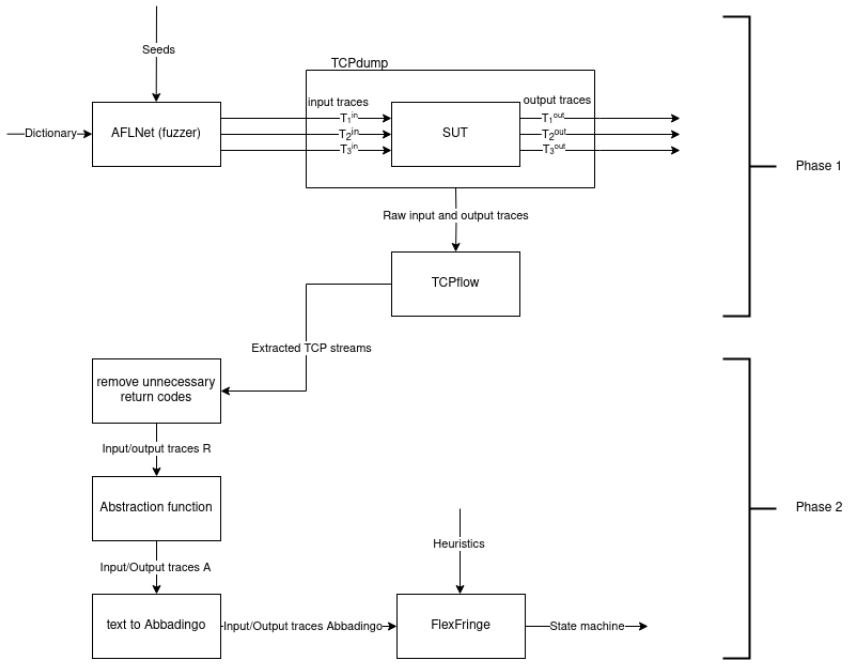


Figure 5.1: Process of generating state machines with FlexFringe using AFLNet (fuzzer).

installation process. In the Dockerfile from Appendix A, the Ubuntu version is specified to 18.04, the required packages and repositories are downloaded (LightFTP, Bftpd, and ProFuzzBench), essential environment variables are configured, and all files are compiled.

To ensure that the error explained in section 5.1.3 does not occur later in the installation process, the container must be started in privileged mode. Starting a container in privileged mode can be done as shown in Listing 5.1. Listing 5.2 demonstrates how to grant executable permissions to the shell script and execute it to launch a Docker container shell.

```
#!/bin/sh

docker run --privileged --name NAME -d -i -t CONTAINER_ID /bin/sh
docker exec -it NAME /bin/sh
```

Listing 5.1: Script to start dockercontainer in privileged mode.

```
    $ sudo chmod u+x FILE_NAME.sh
    $ ./FILE_NAME.sh
```

Listing 5.2: Commands to make the shell script executeable and run it.

5.1.2 Prerequisites AFLNet

A network protocol is a set of rules or conventions that dictate how data should be transmitted, received, and processed in a network. AFLNet uses an initially recorded set of messages of the protocol to build upon and use as initial seed input.

The seed input for LightFTP and Bftpd is the basic input to get past the authentication phase. The seed input starts with FTP commands USER and PASS. These commands are for authentication. The seed input always ends with the FTP command QUIT to not get a timeout. An example seed input is given in Listing 5.3. Some other example seed inputs are given in the LightFTP tutorial folder[2] inside the AFLNet GitHub repository.

```
1  USER ubuntu
2  PASS ubuntu
3  SYST
4  PWD
5  QUIT
```

Listing 5.3: Example seed input for FTP.

Furthermore, a dictionary with all FTP commands, the fftp config file, and a clean-up script need to be provided before fuzzing is possible. All these files can be found in the AFLNet LightFTP tutorial repository folder [3]. For Bftpd these prerequisites are all included in the ProFuzzBench [17] GitHub repository [4]. ProFuzzBench is a benchmarking tool for stateful protocols. This repository includes the clean-up script and the syntax for the config file. To keep the fuzzing of both implementations consistent, the seed input from the LightFTP tutorial folder is used for both implementations.

5.1.3 Solving errors AFLNet

Starting AFLNet's fuzzing process for the first time most likely results in one or both errors. The error given in Figure 5.2 can be resolved by executing the command provided by AFLNet in combination with a root bash script, given in Listing 5.4. The **sudo bash -c** portion is essential, as, without it, the execution is not possible. This is because sudo runs echo as root, but the redirection occurs in the shell with no privileges[26].

```
1 sudo bash -c 'echo core > /proc/sys/kernel/core_pattern'
```

Listing 5.4: ALFNet fuzz command.

[2]https://github.com/aflnet/aflnet/tree/master/tutorials/lightftp/in-ftp
[3]https://github.com/aflnet/aflnet/tree/master/tutorials/lightftp
[4]https://github.com/profuzzbench/profuzzbench/tree/master/subjects/FTP/bftpd

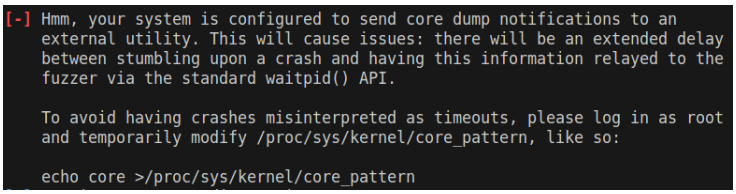


Figure 5.2: Send core dump notification to external utility error.

When attempting to fuzz any protocol for the first time, an error may arise (Figure 5.3). This issue can be easily resolved by including the flag **-m none** in the AFLNet command. This flag will set the max memory size to none.

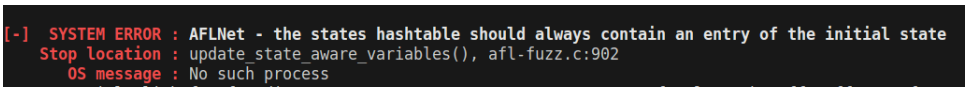


Figure 5.3: Need entry of initial state error.

5.1.4 Fuzzing LightFTP

Now that all prerequisites are done, the fuzzing process can start. In Listing 5.5 the command used to start the fuzzing process is stated for LightFTP.

```
1    afl-fuzz -t 1000+ -d -i $AFLNET/tutorials/lightftp/in-ftp -o out-lightftp -m
      none -N tcp://127.0.0.1/2200 -x $AFLNET/tutorials/lightftp/ftp.dict -P FTP -D
       10000 -q 3 -s 3 -E -R -c ./ftpclean.sh ./fftp fftp.conf 2200
```

Listing 5.5: Starting AFLNet for LightFTP.

The seed input, an output file, the FTP dictionary, the clean-up script, and the config file are all specified. Running the command from Listing 5.5 now starts the fuzzer on port 2200. When done correctly, the status screen of Figure 5.4 will appear in your terminal.

5.1.5 Fuzzing Bftpd

Fuzzing Bftpd is similar to LightFTP (section 5.1.4). The same seed input is used, trying to keep the config also the same. The only difference in the config is the syntax. The syntax for the config is derived from the ProFuzzBench GitHub repository [5]. Executing the command from Listing 5.6 starts fuzzing Bftpd. The status screen from Figure 5.4 will appear.

```
1    afl-fuzz -t 1000+ -d -i $PROFUZZ/in-ftp -o $PROFUZZ/out-ftp/ -m none -N tcp
     ://127.0.0.1/2200 -x $PROFUZZ/ftp.dict -P FTP -D 10000 -q 3 -s 3 -R -c
     $PROFUZZ/clean.sh /home/ubuntu/bdfpd/./bftpd $PROFUZZ/basic.conf 2200
```

Listing 5.6: "Starting AFLNet for Bftpd.

[5] https://github.com/profuzzbench/profuzzbench/blob/master/subjects/FTP/bftpd/basic.conf

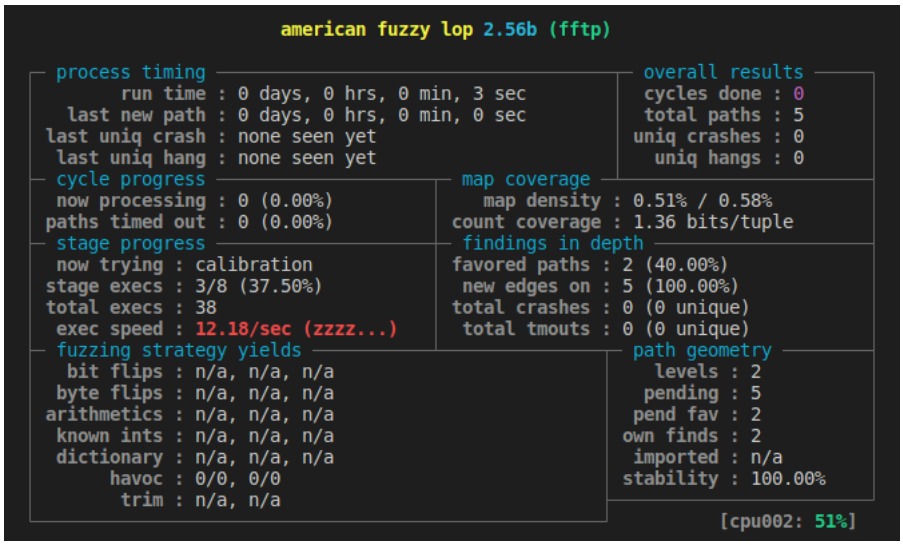


Figure 5.4: AFLNet status screen.

5.1.6 Getting traces

To obtain the traces, all the network traffic will be captured with the use of TCPdump[15]. TCPdump stores the network traffic in a pcap file (Listing 5.7), which can be read with the use of Wireshark[6]. Capturing the traffic is done inside the Docker container. For generating the state machine, the TCP streams are needed. The TCP streams can be extracted from all the traffic with the use of TCPflow[4] as seen in Listing 5.8. TCPflow is an open-source program that extracts TCP data so that it is easy to use for debugging or analysis. For every TCP stream, two separate files are created. The first file contains all the traffic towards the SUT (input trace), and the second file all the output traffic of the SUT (output trace). These files are loaded into the abstraction functions.

```
1    sudo tcpdump -i lo -w output.pcap
```

Listing 5.7: Starting TCPdump and writing to pcap format.

```
1    tcpflow -r output.pcap
```

Listing 5.8: Extracting the TCP streams from a pcap file with TCPflow.

[6]https://www.wireshark.org/

5.2 Generating state machine using FlexFringe

In this section, the process of generating state machines from the obtained traces in section 5.1.6 is given. This section corresponds to phase 2 from Figure 5.1. Firstly, FlexFringe is further explained in section 5.2.1, including which merge heuristics exist. Then a simplified process of transforming the TCP streams captured using TCPdump to Abbadingo format and creating the state machines is shown (section 5.2.2). In section 5.2.3, some problems that occur during the transforming phase are listed. Section 5.2.4 gives the terminology and notations needed for understanding the experiments in section 5.2.5.

5.2.1 FlexFringe

FlexFringe [27] is an open-source passive finite state machine learner as explained in section 2.2.2. FlexFringe uses Abbadingo formatted traces as input to learn the state machines. In Listing 5.9 the Abbadingo format for FlexFringe is given. In the first line, the total number of traces and the length of the alphabet is given. The following lines are the traces. The traces are built as follows: A label, the length of the trace, and then all the symbols.

```
1    # Number of traces # Lenght of alphabet
2    label (lenght n) symbol1 symbol2 ... symboln
```

Listing 5.9: Abbadingo format used for input for FlexFringe.

Merge heuristics

FlexFringe uses heuristics to decide which states can be merged, which must be done in a consistent pattern [23]. Some well-known heuristics have already been added to the FlexFringe repository. These heuristics are stored in an initialization (ini) file. This heuristic file will be passed to FlexFringe the moment a state machine must be created. Some popular heuristics are:

- Markov-chain: Merging states based on their Markov property.
- EDSM: Merging states based on evidence and which is most likely to lead to the target DFA [13].
- Overlap: Merging states based on the overlapping outgoing transitions [23].
- Likelihood: Merging states based on how likely it is according to the log-likelihood [28].
- Mealy: Merging states based on input and output patterns. Outputs are determined by both the input and the current state.

5.2.2 Converting traces to Abbadingo format and creating state machines with FlexFringe

Converting the traces from the two trace files to Abbadingo format is quite challenging. The input and output traces obtained in section 5.1.6 are stripped of any unnecessary response codes (section 5.2.3) and parsed to the abstraction function. The abstraction function eliminates all textual content, providing input traces with only FTP commands and output traces with corresponding response codes. Pairs of inputs and outputs are made, where the output corresponds to the respective input. These pairs are then formatted according to the Abbadingo format outlined in section 5.2.1.

The Abbadingo file can be read by FlexFringe. To create the state machine, the commands from Listing 5.10 and 5.11 need to be executed in order.

```
./flexfringe --ini ini/mealy.ini abbadingo.dat
```

Listing 5.10: Command to run FlexFringe.

```
dot -Tpdf abbadingo.dat.ff.final.dot -o abbadingo-out.pdf
```

Listing 5.11: Command to make pdf from the .dot file.

5.2.3 Problem with double inputs/responses

The Abbadingo format requires that the length of an input trace is the same as the length of an output trace. The traces consist of input and output pairs, which means that when the input trace is not the same length as the output trace, the trace is removed. This may generate inconsistencies. To prevent this from occurring, there are three response codes in the output traces that are removed beforehand. These response codes are not needed. The following response codes are deleted:

Response code 150 "File status okay; about to open data connection." This response code is thrown right before another response. It indicates that the client might wait a moment before the connection is opened, hence FTP gives the 150 code.

Response code 214 "Help message. Explains how to use the server or the meaning of a particular non-standard command. This reply is useful only to the human user." As the definition already explains, it is a reply to help humans understand the non-standard reply.

Response code 220 "Service ready for new user." At the start of every conversation, this response code is thrown to let the client know that the server is ready to communicate.

5.2.4 Terminology and notations

The terminology clarifies the frequently used terms in the experiments (section 5.2.5). The notations employed in the experiments (section 5.2.5) serve the purpose of distinguishing between various abstraction functions and input/output traces.

Terminology

- Abstraction function: Function that gets a separate input and output trace (section 5.1.6), like example trace T_1^{in} and T_1^{out}. This trace is stripped of any unnecessary information (like 220 return codes and text) resulting in $T_1^{in-stripped}$ and $T_1^{out-stripped}$.
- Heuristic: A term used to specify the chosen merge algorithm (section 5.2.1) and includes parameters that dictate the visual appearance of the state machine.
- ParentSizeThreshold: A parameter within the heuristic that denotes the number of transitions required before the child state is visualized in the state machine.

Notations

AF_1: Abstraction function 1 (Appendix B) only uses output traces (like example trace T_1^{out}) to create the input file for FlexFringe. This function removes all text so that only the server response codes remain ($T_1^{out-stripped}$).

AF_2: Abstraction function 2 (Appendix C) uses both the input (like example trace T_1^{in}) and output traces (like example trace T_1^{out}) and creates input-output pairs. The text is removed, leaving only the FTP commands for the input and the server response codes for the output. The FTP commands are translated to a number using a dictionary. Response codes *150, 214* and *220* are removed (section 5.2.3) to avoid more output messages than input messages.

AF_3: Abstraction function 3 (Appendix D) uses both the input (like example trace T_1^{in}) and output traces (like example trace T_1^{out}) and creates input-output pairs. The text is removed, leaving only the FTP commands for the input ($T_1^{in-stripped}$) and the server response codes for the output ($T_1^{out-stripped}$). Authentication commands (USER and PASS) contain a parameter to check if the correct username or password is given. Response codes *150, 214* and *220* are removed (section 5.2.3) to avoid more output messages than input messages.

T_1: Example trace 1 in pcap format, lines starting with a digit are the output messages:

```
220 LightFTP server v2.0a ready
USER anonymous
331 User anonymous OK. Password required
PASS ubuntu
230 User logged in, proceed.
SYST
215 UNIX Type: L8
PWD
257 "/" is a current directory.
PORT 127,0,0,1,152,193
200 Command okay.
LIST
150 File status okay; about to open data connection.
451 Requested action aborted. Local error in processing
.
QUIT
221 Goodbye!

```

T_1^{in}: Example input trace T_1^{in}:

```
USER anonymous
PASS ubuntu
SYST
PWD
PORT 127,0,0,1,152,193
LIST
QUIT

```

T_1^{out}: Example output trace T_1^{out}:

```
220 LightFTP server v2.0a ready
331 User anonymous OK. Password required
230 User logged in, proceed.
215 UNIX Type: L8
257 "/" is a current directory.
200 Command okay.
150 File status okay; about to open data connection.
451 Requested action aborted. Local error in processing
.
221 Goodbye!

```

$T_1^{in-stripped}$: An example of the input trace T_1 after removing all text for abstraction function AF_3. USER and PASS have a parameter:

```
USER anonymous
PASS ubuntu
SYST
PWD
```

```
    PORT
    LIST
    QUIT

```

$T_1^{out-stripped}$: An example of the output trace T_1 after removing all text and unnecessary response codes for abstraction function AF_3:

```
    331
    230
    215
    257
    200
    451
    221

```

T_2: Example trace T_2 in pcap, lines starting with a digit are output messages:

```
    220 LightFTP server v2.0a ready
    .@iic*.R.^YST.@iiR*.R
    SYSTPASS ubuntu
    501 Syntax error in parameters or arguments.
    SYST
    215 UNIX Type: L8
    PWD
    530 Please login with USER and PASS.
    PORT 127,0,0,1,152,193
    530 Please login with USER and PASS.
    LIST
    530 Please login with USER and PASS.
    QUIT
    221 Goodbye!
```

5.2.5 Initial experiments

In this subsection, all experiments conducted to derive the final state machine are presented in chronological order. These experiments aim to explore different abstraction functions and heuristics to identify a suitable state machine for FTP. The ParentSizeThreshold was not taken into consideration for the experiments since the input and output sets lacked a sufficient number of traces for the state machine to become complex or messy. Experiment 4 uses the Mealy heuristic in combination with an abstraction function that uses both input and output. That combination works best for generating a state machine.

Experiment 1

Abstraction function: AF_1 (section 5.2.4)

Heuristic: Markov chain

Set: S_1: Output traces obtained by fuzzing LightFTP for 1,5 minutes

Abbadingo:

```
1 530
220 8 331 230 215 257 200 150 451 221

```

Listing 5.12: Abbadingo format of t_1 experiment 1.

For the first experiment, only the output traces (like example trace T_1^{out}) are used to generate a state machine. Listing 5.12 provides the Abbadingo format of the example trace T_1^{out} used in this experiment.

The abstraction function used to transform the set S_1 is denoted as AF_1. The text that is included in the traces is removed, resulting in $T_1^{out-stripped}$. Subsequently, the traces are transformed into an Abbadingo file, and by using the Markov merge heuristics, the state machine depicted in Figure 5.5 is generated.

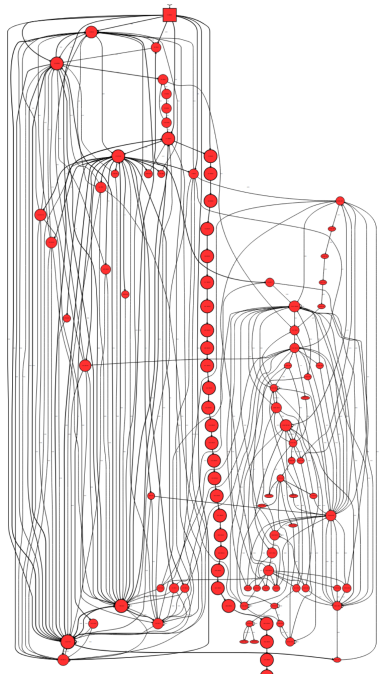


Figure 5.5: The state machine generated for experiment 1. Using the initial set of traces S_1 and an abstraction function AF_1 that only uses the output FTP server response codes. Markov-chain heuristics are applied.

Experiment 2

Abstraction function: AF_2 (section 5.2.4)

Heuristic: Markov chain

Set: S_1: Input/output traces obtained by fuzzing LightFTP for 1,5 minutes

Abbadingo:

```
1 200
−1/220 8 0/331 13/230 19/215 3/257 5/200 6/150 1/221

```

Listing 5.13: Abbadingo format of t_1 experiment 2.

In addition to the previous experiment, this experiment also uses the input. Abstraction function AF_2 translates the FTP command to integers following a standard dictionary. Unnecessary response codes are removed as mentioned in section 5.2.3. The Abbadingo format used for this experiment for example trace T_1 is given in Listing 5.13.

TCPflow creates two files, a server-client conversation file and a client-server conversation file. To create the Abbadingo format, the content of the two files is needed. The function *swap_file_name* returns the name of the other conversation. When ports are reused, the name of the file will append a $c1$, so that it can be distinguished from the other conversation. The *remove_unnecessary_response_codes* will remove all unnecessary response codes as mentioned in section 5.2.3. Another difficulty with the way TCPflow creates files is that sometimes there are more input traces than output traces. This creates irregularities in the Abbadingo format file. By stripping the traces that do not have the same amount of input and output traces, this is solved. An example of a trace that has this problem is an example trace T_2. In trace T_2 it can be seen that there are two inputs before return code 501 is given.

Transforming the set S_1 with the abstraction function AF_2, gives the state machine of Figure 5.6. This state machine is generated with the Markov-chain heuristics.

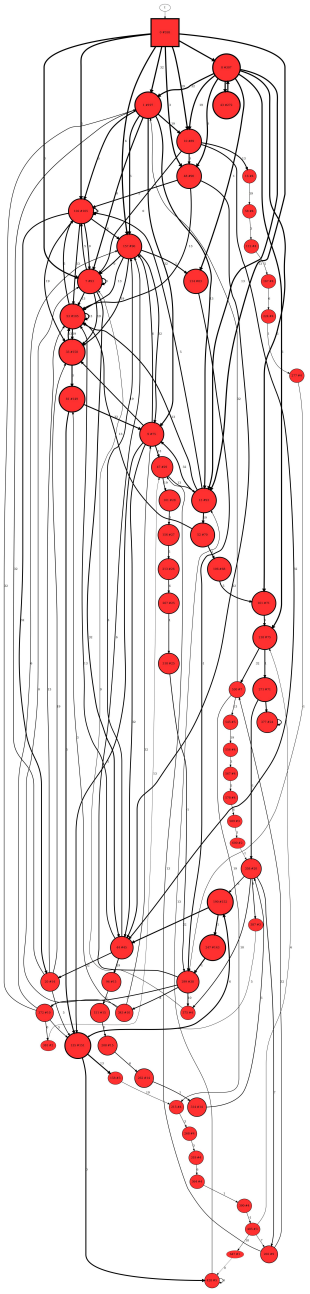


Figure 5.6: The state machine generated for experiment 2. Using the initial set of traces S_1 with an abstraction function AF_2 that uses both input and output traces. Markov-chain heuristics are applied.

Experiment 3

Abstraction function: AF_2 (section 5.2.4)

Heuristic: spdfa

Set: S_1: Input/output traces obtained by fuzzing LightFTP for 1,5 minutes

Experiment 2 used the Markov-chain heuristics as input. By experimenting with different heuristics, the spdfa heuristics generated a state machine that merged the states differently. Using spdfa resulted in Figure 5.7. The state machine is generated on the data from set S_1.

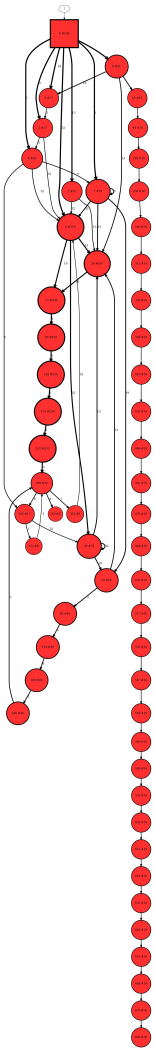


Figure 5.7: The state machine generated for experiment 3. Used initial set of traces S_1 with an abstraction function that uses both input and output traces. Spdfa heuristics are applied.

Experiment 4

Abstraction function: AF_3 (section 5.2.4)

Heuristic: Mealy

Set: S_1: Input/output traces obtained by fuzzing LightFTP for 1,5 minutes

Abbadingo:

```
1 17
1 7 USER/331 PASS/230 SYST/215 PWD/257 PORT/200 LIST
/451 QUIT/221

```

Listing 5.14: Abbadingo format of trace T_1 for experiment 4.

For experiment 4 the abstraction function has changed a lot. Now instead of using a dictionary to translate the commands of the input traces to numbers, the commands themselves are used. How the Abbadingo input for trace T_1 is constructed is shown in Listing 5.14. This gives more clarity to the state machines. After a call with Sicco Verwer, author [27] and co-programmer of FlexFringe, we came to the understanding that the master branch of the FlexFringe repository did not print the return codes of the output traces in the state machines using Mealy heuristics. Previous experiments might have yielded different results had this information been known in advance. A Mealy machine has an input and an output for every transition because the output of the SUT is determined by the input and the current state. A state machine for FTP does need that too. Using initial set S_1 with abstraction function AF_3 and the Mealy heuristic, the state machine of Figure 5.8 is constructed.

In Figure 5.8, a square representing the starting state is accompanied by circles of varying sizes. The size of the circles corresponds to the amount of incoming transitions. Larger circles indicate a higher number of incoming transitions.

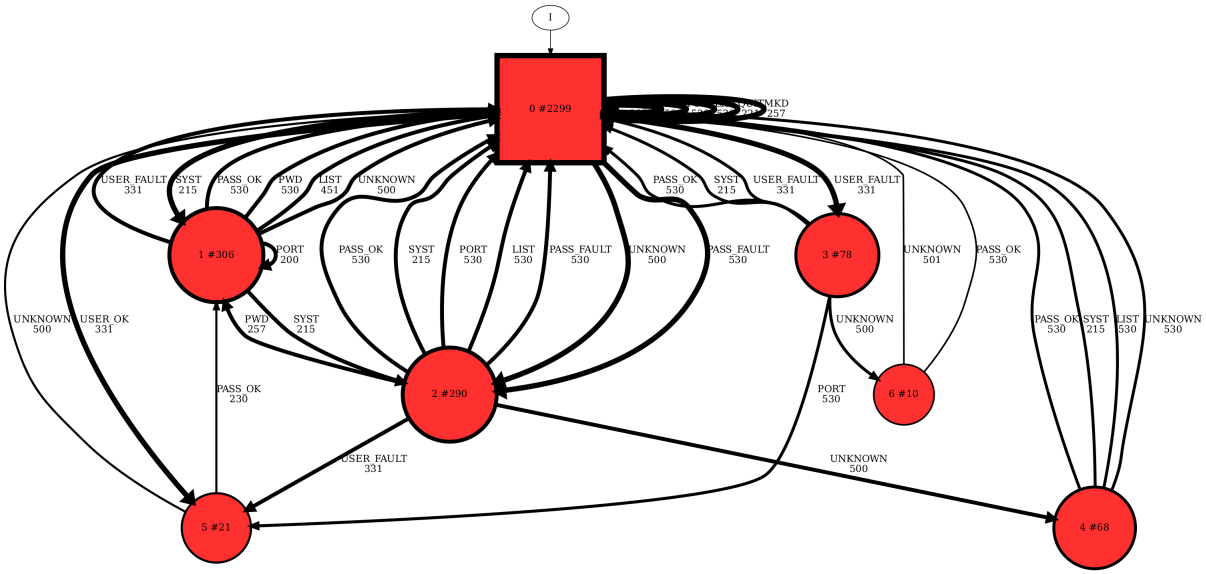


Figure 5.8: The state machine generated for experiment 4. Used initial set S_1 with an abstraction function AF_3. Mealy heuristics are applied.

Chapter 6

Differential fuzzing of two FTP server implementations

In this chapter, two server implementations of the FTP protocol will be compared. Firstly, the parameters used for generating the state machines are given in section 6.1. Subsequently, the state machine of LightFTP is given in section 6.2. In section 6.3, the state machine of Bftpd is provided. Finally, in section 6.4, the two state machines are compared to identify differences.

6.1 Parameters for generating state machines

In this section, the parameters discovered in the initial experiments (section 5.2.5) are explained. First, the heuristic is explained in more detail. Subsequently, the process of the abstraction function is discussed. Lastly, the purpose of the ParentSizeThreshold is explained.

6.1.1 Heuristic

The heuristic discovered during the experiments is the Mealy heuristic. This heuristic utilizes both input and output traces to merge states (see section 5.2.1). The server implementations of the protocols produce outputs that are influenced by both the given input and their current state. Therefore, the Mealy heuristic is the appropriate choice for generating the state machines of the two server implementations.

6.1.2 Abstraction function

The experiments showed that the abstraction function AF_3 (section 5.2.4) worked best. Initially, the *150, 214*, and *220* response codes are removed from the output traces to prevent the occurrence of the double response problem (section 5.2.3). Then, the function pairs the input-output traces,

ensuring that the input messages triggering the output responses are correctly matched. All textual content is removed from the traces. An example is given in section 5.2.4, where the input trace T_1^{in} is abstracted to $T_1^{in-stripped}$ and, similarly, for T_1^{out}, the abstracted version is denoted as $T_1^{out-stripped}$. The authentication commands (USER and PASS) have a parameter and will be checked if the username and password are correct. If the username is correct, the FTP command is translated to 'USER_OK'; otherwise, it is translated to 'USER_FAULT'. The same applies to the password, with 'PASS_OK' and 'PASS_FAULT'. Lastly, the input-output pairs are put in Abbadingo format. An example of the Abbadingo format for both implementations is given in Listing 6.1.

```
1 7
1 7 USER_OK/331 PASS_OK/230 SYST/215 PWD/257 PORT/200 LIST
/451 QUIT/221

```

Listing 6.1: Abbadingo format of example trace T_1.

6.1.3 ParentSizeThreshold

The ParentSizeThreshold is a threshold specifying the minimum number of incoming transitions required for visualization in the state machine (section 5.2.4). This threshold is employed during the learning process and is solely used to generate visually appealing state machines. The default threshold (ParentSizeThreshold=-1) is employed in the experiments. For the full state machines (see Appendix E) the default threshold is used.

6.2 State machine of LightFTP

Version: LightFTP v2.3

Abstraction function: AF_3 (sections 5.2.4 and 6.1.2)

Heuristic: Mealy (section 6.1.1) with ParentSizeThreshold (section 6.1.3) = 150

Set: S_2: Input/output traces obtained by fuzzing LightFTP for 60 minutes

To generate the state machine of LightFTP (Figure 6.1), the process outlined in phase 2, as depicted in Figure 5.1, is applied to set S_2. S_2 is obtained by running AFLNet on LightFTP for 60 minutes, which resulted in 3939 traces. Abstraction function AF_3 is used to generate the Abbadingo file used as input for FlexFringe. AF_3 is discussed in section 6.1.2. The Mealy heuristic is used with a ParentSizeThreshold of 150, resulting in the state machine of Figure 6.1.

Appendix E contains the state machine of LightFTP with the default ParentSizeThreshold specified. This state machine has 83 states.

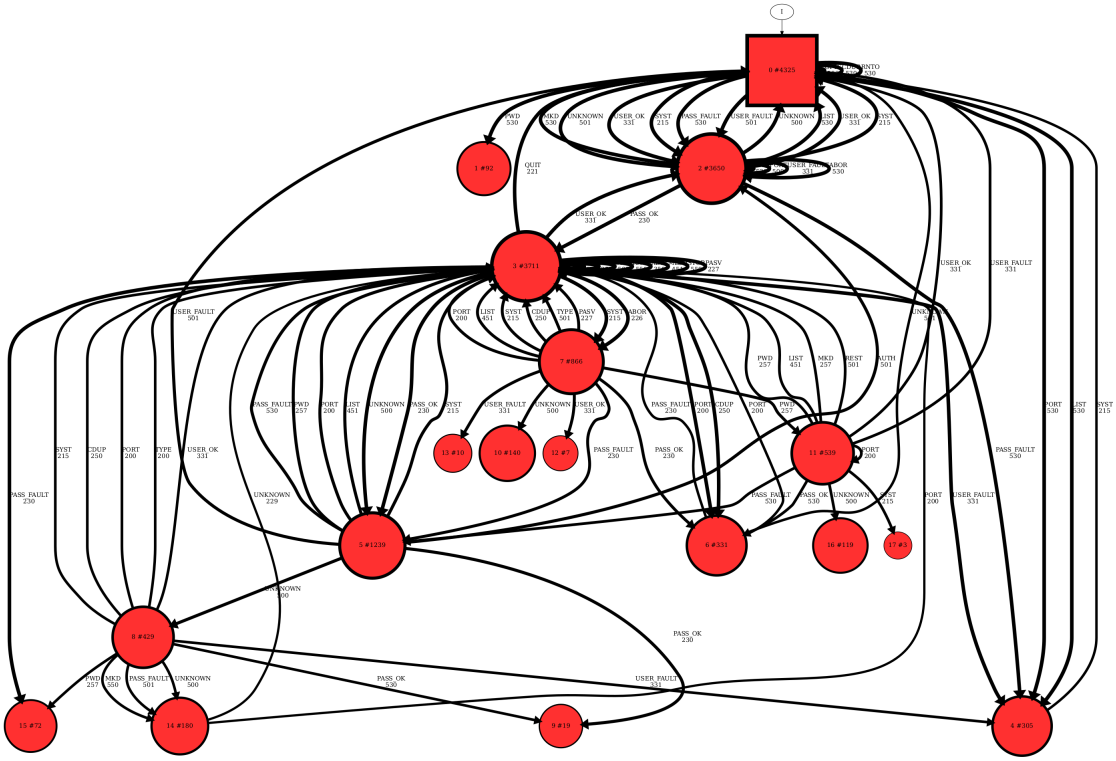


Figure 6.1: State machine of LightFTP generated with set of traces S_2, abstraction function AF_3 and the Mealy heuristic. ParentSizeThreshold set to 150.

6.3 State machine of Bftpd

Version: Bftpd 5.7

Abstraction function: AF_3 (sections 5.2.4 and 6.1.2)

Heuristic: Mealy (section 6.1.1) with ParentSizeThreshold (section 6.1.3) = 100

Set: S_3: Input/output traces obtained by fuzzing Bftpd for 60 minutes

To generate the state machine for Bftpd, the same steps taken for LightFTP (section 6.2) are replicated with set S_3. Set S_3 is obtained by fuzzing Bftpd for 60 minutes, resulting in 3554 traces. The ParentSizeThreshold is smaller for Bftpd because fuzzing Bftpd for 60 minutes gave fewer traces than LightFTP, this is because Bftpd is slower than LightFTP.

Appendix E contains the state machine of Bftpd with ParentSizeThreshold of 3, because after running the dot command (section 5.2.2) for five hours, still no state machine was generated. Inspecting the file from which

the state machines are generated showed that with the default ParentSizeThreshold Bftpd has 297 states. Choosing a ParentSizeThreshold of 3 reduced the time to visualize the state machine to 30 seconds.

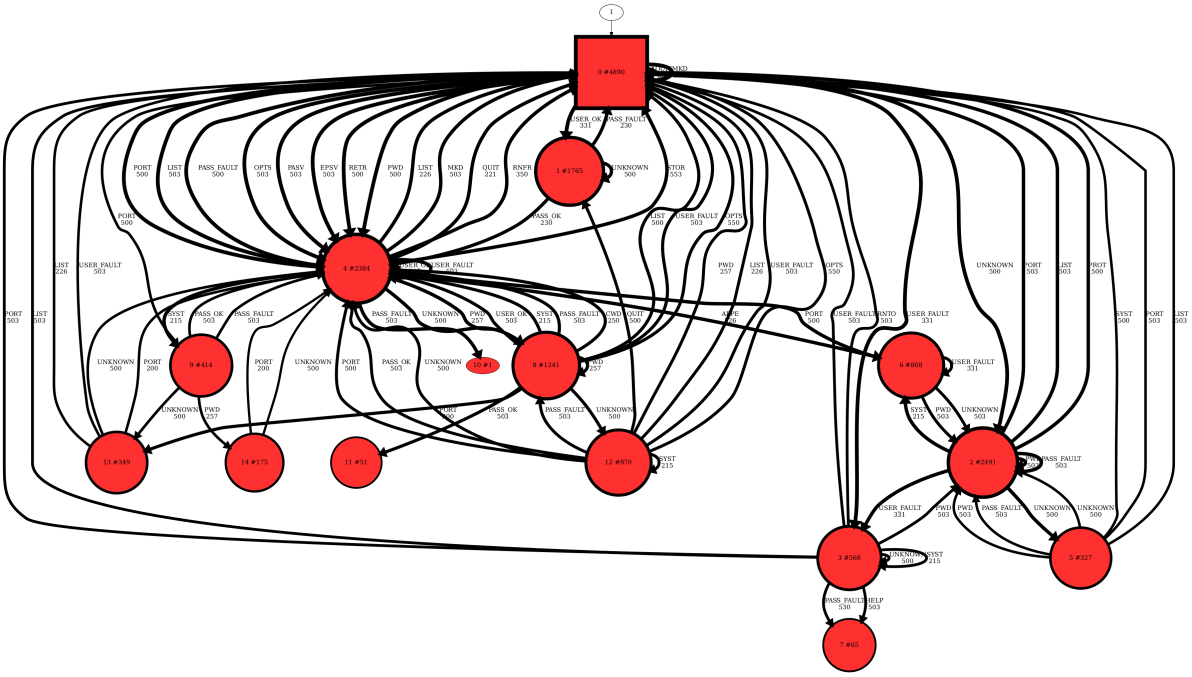


Figure 6.2: State machine of Bftpd generated with set of traces S_3, abstraction function AF_3 and the Mealy heuristic. ParentSizeThreshold set to 100.

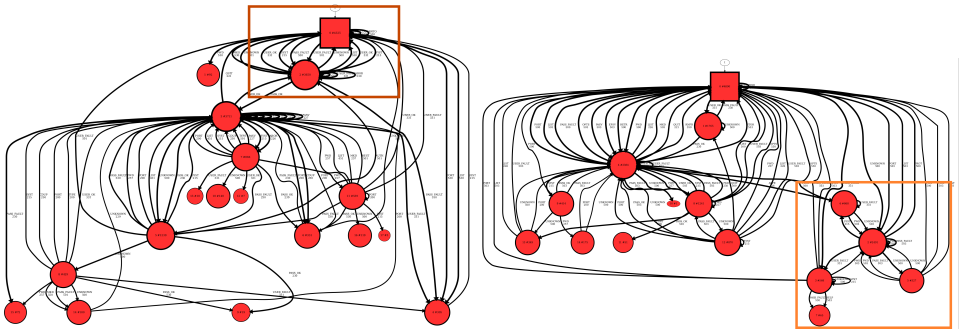
6.4 Comparing LightFTP and Bftpd

In this section, the process of differential fuzzing takes place (as explained in section 2.1.4). Firstly, the notable differences are given (section 6.4.1). Then in section 6.4.2, the possible vulnerabilities are discussed.

6.4.1 Differences in state machines

- A difference between LightFTP and Bftpd is how they approach clients who are not logged in. Bftpd has a separate set of states for those cases. In the graph, these states can be found by incoming transitions with 'USER_FAULT' commands. Figure 6.3 shows the transitions and states in both implementations where failed authentications are handled. The existence of these extra states may indicate an implementation error in Bftpd. It is expected that providing an incorrect username and password will result in reverting to the starting state to attempt authentication again.

A possible explanation could be that, after a certain number of failed login attempts, the client is redirected to another state. However, this behavior is not documented anywhere.



(a) Failed authentication of LightFTP. (b) Failed authentication of Bftpd.

Figure 6.3: Boxed in orange is the part of the state machine where failed authentications are handled.

- The number of states differs between the two implementations. Where LightFTP has only 83 states in the full-state machine (Appendix E), Bftpd has 297 states in the full-state machine (Appendix E).

6.4.2 Possible vulnerabilities

Examining the full-state machines can reveal possible security concerns. This section lists some potential vulnerabilities. Additional research is necessary to find the potential dangers associated with these vulnerabilities (section 7.4).

2xx A response code beginning with "2xx" corresponds to the successful completion of a requested command. The response codes *230* and *232* are issued upon a successful login. When a client provides the correct password with previously a correct username, the server responds with a *230* code. The same applies when a client provides a correct username along with some correct security data, then the *232* code is thrown.

Security issues arise for these response codes when they are issued in response to an unknown FTP command or an incorrect password provided by the client. Such behavior might indicate potential vulnerabilities, suggesting that other inputs could potentially bypass the server's authentication mechanism. Table 6.1 provides three command-response combinations associated with "2xx" responses. LightFTP does not have any unknown commands that result in a successful login for ei-

ther the *230* or the *232* codes. In contrast, Bftpd has two unknown input traces that result in a *230* response code.

In instances where an incorrect password is entered but results in a successful login, LightFTP has nine occurrences, while Bftpd has three. However, upon reviewing the Abbadingo file and investigating these combinations, it was revealed that for most, if not all, cases, this combination was triggered after an already successful login, resulting in false positives. This mitigates the security concern.

3xx A response code beginning with "3xx" corresponds to an accepted command, but a subsequent command with additional information still needs to be provided. The response codes *331* and *336* are issued upon an existing username. The distinction between these two codes lies in the fact that for *331*, a password is required for the specified user, while for *336*, a challenge must be answered instead.

Security issues arise for these response codes when they are issued in response to an unknown FTP command provided by the client. Such behavior suggests that a command other than 'USER' results in the recognition of an existing user. Table 6.1 provides two command-response combinations associated with "3xx" responses. LightFTP has one unknown command leading to a *331* response code, while Bftpd has two such instances. For the *336* code, neither implementation has an unknown command that leads to that specific response code.

Table 6.1 enumerates the number of transitions associated with potentially insecure command-response combinations. Additional research is required to determine whether these combinations represent security vulnerabilities (section 7.4).

Command-response combination	LightFTP	Bftpd
UNKNOWN/230	0	2
PASS_FAULT/230	9	3
UNKNOWN/232	0	0
UNKNOWN/331	1	2
UNKNOWN/336	0	0

Table 6.1: Number of potential insecure transitions for some command-response code combinations.

Chapter 7

Future work

In this chapter, the possible extensions to differential fuzzing are discussed. In section 7.1, the usage of a different fuzzer is discussed. In section 7.2 it is discussed that the same research can be done on a different protocol. Then, in section 7.3, generating state machines with a different learner is discussed. Lastly, in section 7.4, it is discussed that more research needs to be done on the possible security risks.

7.1 Using a different fuzzer

In section 2.1, different types of fuzzers are listed. In the future, the same research can be done with the use of a different fuzzer. A suitable fuzzer would be BooFuzz [18] as discussed in section 3.2. For this research, AFLNET was chosen. AFLNet is a greybox fuzzer (section 2.1.3), whereas BooFuzz is a blackbox fuzzer (section 2.1.1). It can be insightful to compare the outcomes generated by a blackbox fuzzer with those of a greybox fuzzer.

7.2 Differential fuzzing of a different protocol

In Chapter 4, different protocols are listed. For this study, the FTP protocol has been selected. The research conducted in this study can be applied to stateful protocols that have multiple server implementations. An example of such a viable protocol is presented in section 4.2, namely the Transmission Control Protocol (TCP [1]). TCP is a connection-oriented protocol, meaning it maintains state information. This characteristic classifies it as a stateful protocol, making it well-suited for future research.

7.3 Generating state machines with a different learner

As outlined in section 2.2, there are two types of learners: passive and active. In this study, a passive automaton learner FlexFringe [27] is used to

generate a state machine based on the fuzzed traces. An active approach to this research has already been done by Cristian Daniele [1]. He generated state machines of different FTP implementations using the active automaton learner Learnlib [20]. Apart from these two learning algorithms, future research could explore similar investigations with various passive or active learning algorithms.

7.4 Examining the possible vulnerabilities

The potential vulnerabilities discussed in section 6.4.2 require further investigation. For this, it is necessary to understand the specific client input traces that prompted the server to return *230, 331, 336* response codes. Analyzing these input traces will provide insights into how the interaction came together and help determine whether the observed response codes indicate security vulnerabilities or if they are the result of normal FTP behavior. Identifying potentially insecure prompts can be achieved using the information gathered in this study, but a significant modification to the abstraction function is needed. Another approach to investigating security risks is to conduct fuzzing (as described in section 5.1) again, but this time without providing the correct username and password in the seed input. If any portion of the state machine appears to be situated after authentication, it suggests that the authentication process might be bypassed. Unfortunately, due to time constraints, these tasks could not be carried out within the scope of this thesis.

[1] https://github.com/cristiandaniele/ftp-statemodel-learner

Chapter 8

Conclusions

In this thesis, we worked towards finding a way to do differential fuzzing of a stateful protocol using FlexFringe. By fuzzing two FTP server implementations using AFLNet (see section 5.1), a substantial number of traces are collected. These traces are preprocessed and formatted to be compatible with FlexFringe to generate state machines (section 5.2). By performing differential fuzzing on both state machines (section 6.4), it was identified that there is a difference in the number of states between the two implementations and how they deal with authentication (section 6.4.1). Furthermore, by closely inspecting the state machines for vulnerabilities, some inputs that have the potential to be a security risk were uncovered (see section 6.4.2).

- **Setup:** In section 5.1.1, setting up AFLNet using the Dockerfile was a simple task. Starting the docker container in privileged mode is a must, otherwise, errors will appear in later stages. Do not forget to patch the GitHub repository, or else you have a hard time debugging errors. Setting up FlexFringe is just as simple as AFLNet. Following the instructions on the GitHub repository, the software is installed in a matter of minutes.
- **Fuzzing and collecting traces:** Fuzzing both LightFTP and Bftpd did not cause any major problems. After solving the errors from section 5.1.3, which were caused by a non-privileged Docker image, fuzzing could start right away. Getting the traces is done by running a TCPdump inside the same Docker container. Extracting the TCP streams is done by using TCPflow. Unfortunately, TCPflow generates distinct TCP streams for input and output traces, which does not help in making pairs of input and output traces.
- **Experimenting with different factors:** Through experimentation with various factors in section 5.2.5, the parameters for generating state machines for stateful protocols were determined. The used parameters are given in section 6.1. The three factors that influence

the appearance of the state machines are the abstraction function, the heuristics, and the ParentSizeThreshold. The abstraction function takes input-output pairs, eliminates all textual content, and excludes the *150, 214*, and *220* response codes. The Mealy heuristic is employed to ensure that transitions use both inputs and outputs. A ParentSizeThreshold of more than 100 is utilized in sections 6.2 and 6.3. For section 6.4, the default threshold (ParentSizeThreshold = -1) is employed.

- **Results of differential fuzzing:** In Chapter 6, the final state machines for LightFTP and Bftpd are presented. Comparing the two state machines discovered that Bftpd has 297 states, whereas LightFTP has only 83 states. Another difference between implementations is how they handle authentication. LightFTP sets the failed authentications back to the starting state. Bftpd has a small set of states that handle failed authentications. This small set may indicate an implementation error because it is expected that providing an incorrect username and password will result in revering to the starting state to attempt authentication again.

  After inspecting the state machines for both implementations, some command-response code combinations were found that could be potential security risks. In Table 6.1 all combinations and the number of times they occur are listed. These combinations can be vulnerabilities because an unknown command triggers a critical FTP server response code. Further research needs to be done to check if these combinations are security risks (section 7.4).

- **Experience:** Fuzzing a stateful protocol like LightFTP with AFLNet was simple, and the installation went smoothly. AFLNet has outstanding documentation on the GitHub page, and it even has premade tutorials for some common protocols. FlexFringe has an easy installation process, and the first state machines are created quickly. However, understanding the Abbadingo format as input for FlexFringe and how the heuristics influence the state machine took more time to figure out. Documentation for Abbadingo is limited, which made finding the right format challenging, as described in section 5.2.5. The heuristics contain information for merging states during FlexFringe's learning process. It took a considerable amount of time to identify the Mealy heuristic, partly due to the master branch of the FlexFringe GitHub repository did not correctly print the outputs when the Mealy heuristic was selected.

- **What would be done differently next time:** While experimenting with different abstraction functions, heuristics, and Abbadingo outputs, there was no clear structure in trying to figure out what

would work best. All the variables are interdependent, so changing multiple variables at once does not provide any understanding of the single variables. Next time, I would consider changing one variable at a time until it reaches an approximate or desired output.

Bibliography

[1] Transmission Control Protocol. RFC 793 https://www.ietf.org/rfc/rfc793.txt, sept 1981.

[2] File Transfer Protocol. RFC 959 https://www.ietf.org/rfc/rfc959.txt, oct 1985.

[3] Pedram Amini. Sulley. https://github.com/OpenRCE/sulley, 2012.

[4] Jeremy Elson. tcpflow – a TCP flow recorder. https://www.circlemud.org/jelson/software/tcpflow/, 2003.

[5] Robert B. Evans and Alberto Savoia. Differential testing: A new approach to change detection. In *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, ESEC-FSE companion '07, page 549–552, New York, NY, USA, 2007. Association for Computing Machinery.

[6] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. Grayc: Greybox fuzzing of compilers and analysers for c. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 1219–1231, New York, NY, USA, 2023. ACM.

[7] Mark Fijneman. Fuzzing open source OPC UA implementations. Bachelor thesis, Radboud University, 2023.

[8] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43(6):206–215, jun 2008.

[9] hfiref0x. Lightftp. https://github.com/hfiref0x/LightFTP/tree/master, 2015.

[10] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, jul 1976.

[11] Fouad Lamsettef. Extending ProFuzzBench: A benchmark for stateful fuzzers. Bachelor thesis, Radboud University, 2023.

[12] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[13] S.M. Lucas and T.J. Reynolds. Learning dfa: evolution versus evidence driven state merging. In *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.*, volume 1, pages 351–358 Vol.1, 2003.

[14] Alexander Maier. Online passive learning of timed automata for cyber-physical production systems. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*, pages 60–66, 2014.

[15] Steve McCanne, Sally Floyd, van Jacobson, and Vern Paxson. Tcpdump & libpcap. `https://www.tcpdump.org/`, 1988.

[16] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[17] Roberto Natella and Van-Thuan Pham. Profuzzbench: A benchmark for stateful protocol fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[18] Joshua Pereyda. BooFuzz: Network protocol fuzzing for humans. `https://github.com/jtpereyda/boofuzz#boofuzz-network-protocol-fuzzing-for-humans`, 2018.

[19] Pham, Van-Thuan, Böhme, Marcel, Roychoudhury, and Abhik. Aflnet: A greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465, 2020.

[20] Harald Raffelt, Bernhard Steffen, and Therese Berg. Learnlib: A library for automata learning and experimentation. In *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems*, FMICS '05, page 62–71, New York, NY, USA, 2005. Association for Computing Machinery.

[21] Michael Renner, TJ Saunders, and Jesse Sipprell. Proftpd. `http://www.proftpd.org/`, 1999.

[22] Iman Saberi, Fathiyeh Faghih, and Farzad Sobhi Bavil. A passive online technique for learning hybrid automata from input/output traces. *ACM Trans. Embed. Comput. Syst.*, 22(1), oct 2022.

[23] Rafail Skouloss. Learning state machines faster using locality-sensitive hashing and an application in network-based thread detection. Master thesis, National Technical University of Athens in collaboration with TU Delft, 2020.

[24] J.F.R.G. Smith. Bftpd. `https://bftpd.sourceforge.net/contact.html`, 2004.

[25] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient instrumentation for code coverage testing. *SIGSOFT Softw. Eng. Notes*, 27(4):86–96, jul 2002.

[26] Gillis (`https://unix.stackexchange.com/users/885/gilles-so-stop-being-evil`). Why is editing core_pattern restricted? `https://unix.stackexchange.com/questions/343275/why-is-editing-core-pattern-restricted`, 2017.

[27] Sicco Verwer and Christian A. Hammerschmidt. FlexFringe: A passive automaton learning package. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 638–642, 2017.

[28] Sicco Verwer, Cees Witteveen, and Mathijs de Weerdt. A likelihood-ratio test for identifying probabilistic deterministic real-time automata from positive data. In *International Colloquium on Grammatical Inference*, pages 203–216, 2010.

[29] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 511–522. Association for Computing Machinery, 2013.

Appendix A

Dockerfile for setting up AFLNet and lightFTP

Appendix A shows the Dockerfile that is used in section 5.1.1 to set up AFLNet which includes LightFTP.

```
# syntax=docker/dockerfile-upstream:master-labs
FROM ubuntu:18.04

# Install common dependencies
RUN apt-get -y update && \
    apt-get -y install sudo \
    apt-utils \
    build-essential \
    openssl \
    clang \
    graphviz-dev \
    git \
    libcap-dev \
    llvm-dev \
    libgnutls28-dev \
    tcpdump \
    wget

# Add a new user ubuntu, pass: ubuntu
RUN groupadd ubuntu && \
    useradd -rm -d /home/ubuntu -s /bin/bash -g ubuntu -G sudo -
    u 1000 ubuntu -p "$(openssl passwd -1 ubuntu)"

# Use ubuntu as default username
USER ubuntu
WORKDIR /home/ubuntu
# Download and compile AFLNet
ENV LLVM_CONFIG="llvm-config-6.0"


RUN git clone https://github.com/aflnet/aflnet && \
    cd aflnet && \
```

```
32      make clean all && \
33      cd llvm_mode make && make
34
35  # Set up environment variables for AFLNet
36  ENV AFLNET="/home/ubuntu/aflnet"
37  ENV PATH="${PATH}:${AFLNET}"
38  ENV AFL_PATH="${AFLNET}"
39  ENV AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1 \
40      AFL_SKIP_CPUFREQ=1
41  ENV WORKDIR="/home/ubuntu"
42
43  # Download and compile LightFTP
44  RUN cd /home/ubuntu && \
45      git clone https://github.com/hfiref0x/LightFTP.git && \
46      cd LightFTP && \
47      git checkout 5980ea1 && \
48      patch -p1 < ${AFLNET}/tutorials/lightftp/5980ea1.patch && \
49      cd Source/Release && \
50      CC=afl-clang-fast make clean all
51
52
53  # Set up LightFTP for fuzzing
54  RUN cd /home/ubuntu/LightFTP/Source/Release && \
55      cp ${AFLNET}/tutorials/lightftp/fftp.conf ./ && \
56      cp ${AFLNET}/tutorials/lightftp/ftpclean.sh ./ && \
57      cp -r ${AFLNET}/tutorials/lightftp/certificate ~/ && \
58      mkdir ~/ftpshare
59
60  ENV PROFUZZ="/home/ubuntu/profuzzbench/subjects/FTP/BFTPD"
61
62  # Set up BFTPD for fuzzing including profuzzbench
63  RUN cd ${WORKDIR} && \
64      wget https://sourceforge.net/projects/bftpd/files/bftpd/
    bftpd-5.7/bftpd-5.7.tar.gz && \
65      tar -zxvf bftpd-5.7.tar.gz && \
66      git clone https://github.com/profuzzbench/profuzzbench.git
    && \
67      cd bftpd && \
68      patch -p1 < ${PROFUZZ}/fuzzing.patch && \ && \
69      CC="afl-clang-fast" CXX="afl-clang-fast++" ./configure --
    enable-devel=nodaemon:nofork && \
70      AFL_USE_ASAN=1 make $MAKE_OPT
```

Listing A.1: Dockerfile AFLNet, LightFTP, Bftpd and ProFuzzBench

Appendix B

Script for generating Abbadingo file for experiment 1

```python
import glob

path = "YOUR/PATH/HERE/TO/FOLDER/WITH/TRACES"

all_files = glob.glob(f"{path}/*")

number_of_useful_files = 0

#create a new file where the abbadingo file will be written in
with open("abbadingo.dat", "w") as abbadingo:
    for file_path in all_files:
        if file_path.startswith(f"{path}/127.000.000.001.02200"):
            with open(file_path, 'r', encoding='latin1') as file:
                lines = file.readlines()
                first_word = lines[0].split()[0]
                lines.pop(0)
                length = len(lines)

                string = f"{first_word} {length}"
                for line in lines:
                    word = line.split()[0]
                    # Do not add if word is not a number
                    if word.isdigit():
                        string += f" {word}"



                abbadingo.write(string + "\n")
                number_of_useful_files += 1
    abbadingo.seek(0)
    abbadingo.write(f"{number_of_useful_files} 530\n")
```

Listing B.1: Script to transform traces to Abbadingo format

Appendix C

Script for generating Abbadingo file for experiment 2 and 3

```
import glob

def swap_file_name(filename):
    if not filename.endswith("c1"):
        parts = filename.split("-")
        if len(parts) == 2:
            src, dst = parts
            src_ip, src_port = src.split('.',1)
            dst_ip, dst_port = dst.split('.',1)
            new_filename = f"{dst_ip}.{dst_port}-{src_ip}.{src_port}"
            return new_filename
        else:
            return None
    else:
        filename = filename.replace("c1", "")
        parts = filename.split("-")
        if len(parts) == 2:
            src, dst = parts
            src_ip, src_port = src.split('.',1)
            dst_ip, dst_port = dst.split('.',1)
            new_filename = f"{dst_ip}.{dst_port}-{src_ip}.{src_port}c1"
            return new_filename
        else:
            return None

def strip_lines_with_4x(lines):
    new_lines = []
    for line in lines:
        if line.startswith("4"):
            continue
        else:
            new_lines.append(line)
    return new_lines

path = "YOUR/PATH/HERE/TO/FOLDER/WITH/TRACES"

FTP_COMMANDS_DICT = {
    "INITIAL": -1,
    "USER": 0,
    "QUIT": 1,
    "NOOP": 2,
    "PWD": 3,
    "TYPE": 4,
    "PORT": 5,
    "LIST": 6,
    "CDUP": 7,
    "CWD": 8,
    "RETR": 9,
    "ABOR": 10,
    "DELE": 11,
```

```
    "PASV": 12,
    "PASS": 13,
    "REST": 14,
    "SIZE": 15,
    "MKD":  16,
    "RMD":  17,
    "STOR": 18,
    "SYST": 19,
    "FEAT": 20,
    "APPE": 21,
    "RNFR": 22,
    "RNTO": 23,
    "OPTS": 24,
    "MLSD": 25,
    "AUTH": 26,
    "PBSZ": 27,
    "PROT": 28,
    "EPSV": 29,
    "HELP": 30,
    "SITE": 31,
    "UNKNOWN": 32,
}

all_files = glob.glob(f"{path}/*")

number_of_useful_files = 0

#create a new file where the abbadingo file will be written in
with open("abbadingo.dat", "w") as abbadingo:
    for file_path in all_files:
        if file_path.startswith(f"{path}/127.000.000.001.02200"):
            file_name = file_path.split("/")[-1]
            swap_name = swap_file_name(file_name)
            with open(file_path, 'r', encoding='latin1') as file, open(f"{path}/{swap_name}", 'r', encoding='latin1') as swap_file:
                lines = file.readlines()
                swap_lines = swap_file.readlines()
                first_word = lines[0].split()[0]

                lines.pop(0)
                length = len(lines)


                # Remove lines that have 400 code
                strip_lines_with_4x(lines)
                string = f"-1/{first_word} {length}"

                if len(lines) == len(swap_lines):

                    for line, swap_line in zip(lines, swap_lines):
                        word_line = line.split()[0]
                        try:
                            word_swap_line = swap_line.split()[0]
                        except:
                            continue

                        if word_swap_line in FTP_COMMANDS_DICT:
                            swap_number = FTP_COMMANDS_DICT[word_swap_line]
                        else:
                            swap_number = 32
                        # Do not add if word is not a number
                        if word_line.isdigit() and swap_number <= 32:
                            string += f" {swap_number}/{word_line}"
                    abbadingo.write(string + "\n")
                    number_of_useful_files += 1
    abbadingo.seek(0)
    abbadingo.write(f"{number_of_useful_files} 200\n")
```

Listing C.1: Script to transform traces to Abbadingo format for experiment 2

Appendix D

Script for generating Abbadingo files from LightFTP or Bftpd TCP streams

```
import glob
import os

def swap_file_name(filename):
    valid_suffixes = ["c1", "c2", "c3", "c4"]

    if not any(filename.endswith(suffix) for suffix in valid_suffixes):
        parts = filename.split("-")
        if len(parts) == 2:
            src, dst = parts
            src_ip, src_port = src.split('.',1)
            dst_ip, dst_port = dst.split('.',1)
            new_filename = f"{dst_ip}.{dst_port}-{src_ip}.{src_port}"
            return new_filename
        else:
            return None

    for suffix in valid_suffixes:
        if filename.endswith(suffix):
            parts = filename.replace(suffix, "").split("-")
            if len(parts) == 2:
                src, dst = parts
                src_ip, src_port = src.split('.', 1)
                dst_ip, dst_port = dst.split('.', 1)
                new_filename = f"{dst_ip}.{dst_port}-{src_ip}.{src_port}{suffix}"
                return new_filename

    return None


def remove_unnecessary_response_codes(lines, prefixes):
    return [line for line in lines if not any(line.startswith(prefix) for prefix
    in prefixes)]


FTP_COMMANDS_DICT = {
    "INITIAL": -1,
    "USER_OK": 0,
    "USER_FAULT":0,
    "QUIT": 1,
    "NOOP": 2,
    "PWD":  3,
    "TYPE": 4,
    "PORT": 5,
    "LIST": 6,
    "CDUP": 7,
```

```
    "CWD":  8,
    "RETR": 9,
    "ABOR": 10,
    "DELE": 11,
    "PASV": 12,
    "PASS_OK": 13,
    "PASS_FAULT":13,
    "REST": 14,
    "SIZE": 15,
    "MKD":  16,
    "RMD":  17,
    "STOR": 18,
    "SYST": 19,
    "FEAT": 20,
    "APPE": 21,
    "RNFR": 22,
    "RNTO": 23,
    "OPTS": 24,
    "MLSD": 25,
    "AUTH": 26,
    "PBSZ": 27,
    "PROT": 28,
    "EPSV": 29,
    "HELP": 30,
    "SITE": 31,
    "UNKNOWN": 32,
}

def abstraction_function(line, swap_line, isCorrectUser):
    '''
    Strips the input and output traces of text and checks if the username and/or password are correct.
    '''
    try:
        word_line = line.split()[0]

    except:
        word_line = "500"
    try:
        swap_line = swap_line.split()

    except:
        swap_line = ["UNKNOWN", "", ""]

    word_swap_line = "UNKNOWN"

    if len(swap_line) >= 1:
        if swap_line[0] == "USER":
            isCorrectUser = swap_line[1] == "ubuntu" or swap_line[1] == "anonymous" if len(swap_line) > 1 else False
            word_swap_line = "USER_OK" if isCorrectUser else "USER_FAULT"
        elif swap_line[0] == "PASS":
            isCorrectPass = swap_line[1] == "ubuntu" if len(swap_line) > 1 else False
            word_swap_line = "PASS_OK" if isCorrectPass and isCorrectUser else "PASS_FAULT"
        else:
            word_swap_line = swap_line[0]

    if not word_swap_line in FTP_COMMANDS_DICT:
        word_swap_line = "UNKNOWN"

    return word_line, word_swap_line, isCorrectUser

def text_to_abbadingo(word_line, word_swap_line, string, different_combinations, check):
    '''
    Transforms the different client commands of the input trace and the server response of the output trace to abbadingo format.
    '''
    if word_line.isdigit():
        check += 1
        string += f" {word_swap_line}/{word_line}"
        combination = f"{word_swap_line}/{word_line}"
        different_combinations.add(combination)
    return string, different_combinations, check



def main(path):

    all_files = glob.glob(f"{path}/*")

    different_combinations = set()
```

```
124
125     number_of_useful_files = 0
126
127     #create a new file where the abbadingo file will be written in
128     with open("abbadingo_flow.dat", "w") as abbadingo:
129         # Go over every file in the folder
130         for file_path in all_files:
131             if file_path.startswith(f"{path}/127.000.000.001.02200"):
132                 # Store the input and output trace file name
133                 file_name = file_path.split("/")[-1]
134                 swap_name = swap_file_name(file_name)
135                 if os.path.exists(file_path) and os.path.exists(f"{path}/{
    swap_name}"):
136                     with open(file_path, 'r', encoding='latin1') as file, open(f"
    {path}/{swap_name}", 'r', encoding='latin1') as swap_file:
137                         # Read the input and output trace contents and remove the
     unnecessary response codes
138                         lines = file.readlines()
139                         lines = remove_unnecessary_response_codes(lines, ["150",
    "220", "214"])
140                         swap_lines = swap_file.readlines()
141
142
143
144                         if len(lines) == len(swap_lines):
145                             # Add the label and length of the trace for Abbadingo
     format
146                             length = len(swap_lines)
147
148                             string = f"A {length+1}"
149                             # Set variable isCorrectUser, true if the trace
    contains "USER ubuntu"
150                             isCorrectUser = False
151                             # Set variable check, check if the length is the same
     as the amount of input/output pairs
152                             check = 0
153
154                             for line, swap_line in zip(lines, swap_lines):
155                                 word_line, word_swap_line, isCorrectUser =
    abstraction_function(line, swap_line, isCorrectUser)
156                                 string, different_combinations, check =
    text_to_abbadingo(word_line, word_swap_line, string, different_combinations,
    check)
157
158                             if check == length:
159                                 abbadingo.write(string + "\n")
160                                 number_of_useful_files += 1
161         abbadingo.seek(0)
162         length_combinations = len(different_combinations)
163         abbadingo.write(f"{number_of_useful_files} {length_combinations}\n")
164
165 if __name__ == "__main__":
166     path = "/flow"
167     main(path)
```

Listing D.1: Script for generating Abbadingo files from the output that TCPflow gives. Separating the abstraction text to abbadingo and remove unnecessary return codes functions.

Appendix E

Full state machines for LightFTP and Bftpd

E.1 State machine of LightFTP

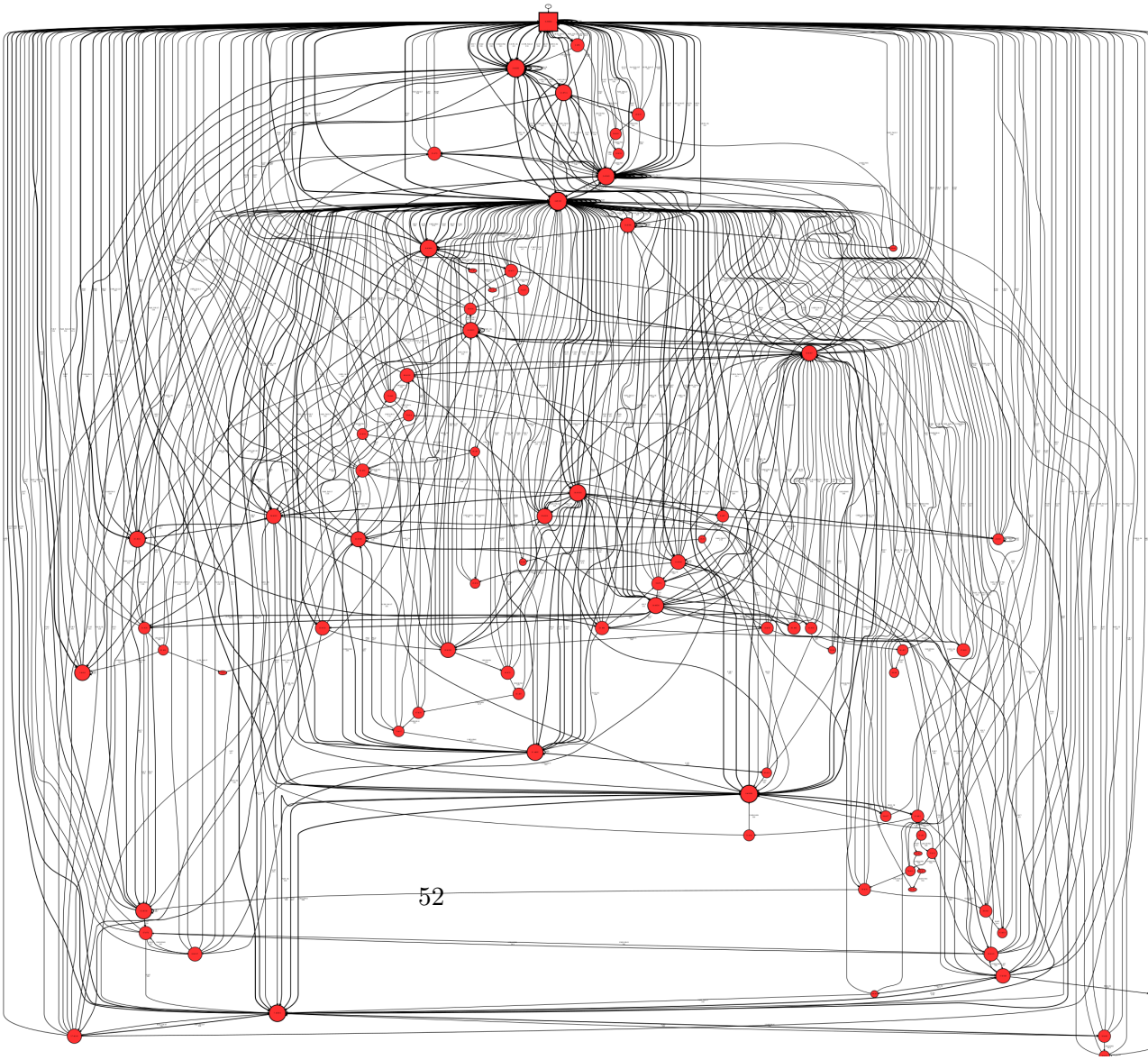


Figure E.1: The full state machine of LightFTP. This state machine has 83 states. The default ParentSizeThreshold is used with abstraction function AF_3 (Appendix D)

E.2 State machine of Bftpd

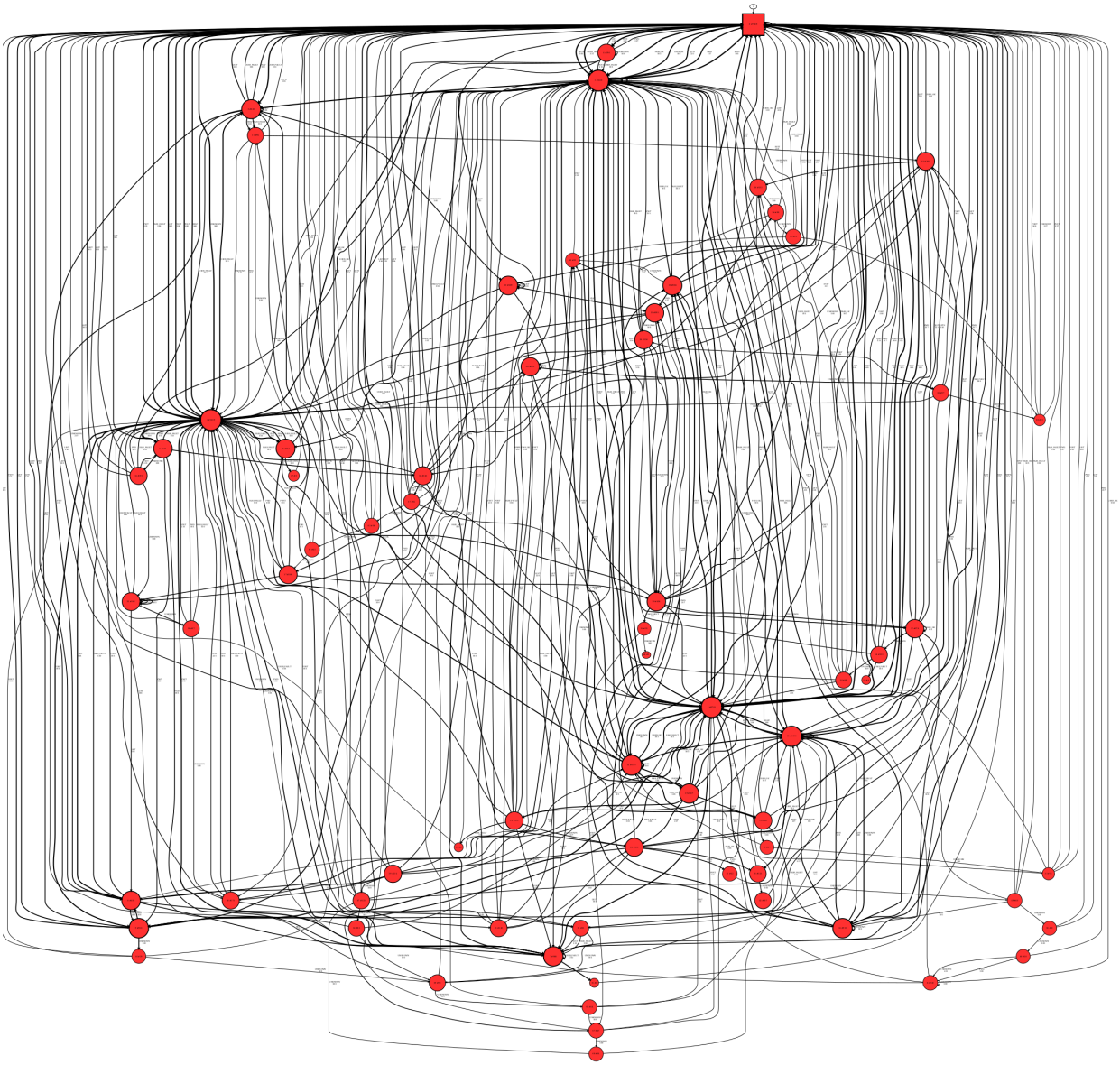


Figure E.2: The full state machine of Bftpd. This state machine has 297 states (not all displayed due to threshold). A ParentSizeThreshold of 3 is used with abstraction function AF_3 (Appendix D)