Bachelor's Thesis Computing Science

Extending ProFuzzBench

A Benchmark for Stateful Fuzzers

Fouad Lamsettef
s1034545

June 28, 2023

First supervisor/assessor:
Dr. Ir. Erik Poll

Second assessor:
Prof. Dr. Frits Vaandrager

Second supervisor:
Cristian Daniele

Third supervisor:
Seyed Bahnam Andarzian

Radboud University

Abstract

Nowadays, fuzzing has become very popular in recent years due to its effectiveness in finding security vulnerabilities in a system under test (**SUT**), which can be easily automated. Fuzzing uses a lot of carefully crafted, automatically generated input and checks whether the SUT performs any unexpected behaviour, like, for example, crashes, which may reveal valuable information for an attacker. Fuzzing is an effective strategy for SUTs that use stateless protocols. However, these fuzzers fall short for SUTs that use stateful protocols because they unable to progress further to a meaningful state in the system. This is mostly the case in network protocols, and the research community has responded by developing a lot of fuzzers which are better suited for stateful systems.

But how does one choose the fuzzer that is the "best" for a specific stateful protocol? To answer that question, we must compare fuzzers with each other for each stateful protocol. This is where ProFuzzBench can offer a solution. ProFuzzBench is a benchmark that provides a framework for testing fuzzers for network protocols. This benchmark includes a handful of open-source network servers for popular network protocols like TLS, SSH, and FTP.

The goal of ProFuzzBench is to provide a framework to benchmark fuzzers in a well-defined environment to obtain reproducible and quantitative results. In this thesis, we extend ProFuzzBench with three fuzzers, namely SNPSFuzzer, Nyx-Net, and BooFuzz and evaluate the process of extending ProFuzzBench. Furthermore, we compare the added fuzzers with AFLnet and AFLnwe, fuzzers which are provided with ProFuzzBench. In the end, we also discuss about the reproducibility of fuzzers.

Contents

Chapter 1

Introduction

Fuzzing is an approach in which a system is fed a large amount of generated inputs to detect security vulnerabilities, such as memory leaks or unexpected crashes. A significant advantage of fuzzing is its ability to find a considerable amount of bugs with relatively little effort. Fuzzers often target **stateless systems** where the **SUT**(system under test) accepts only a single input and the fuzzer attempts numerous inputs to achieve an undesired result[1].

However, some fuzzers struggle with stateful systems because fuzzers can only try to multiple inputs for the initial state and fail to progress to deeper states. This necessitates the use of fuzzers designed to work with stateful systems.

Many network protocols require the system and client connection to be in a state where messages can be exchanged. The state may change between messages sent between the system and client. We refer to these protocols, which require a sequence of messages, as **stateful system**. With stateful systems, each input may alter the internal state of the system.

In this thesis we answer the following questions:

- Is ProFuzzBench a sufficient solution for benchmarking fuzzers for multiple targets like for example FTP and SSH?
- How easy is adding fuzzers to ProFuzzbench?
- Is benchmarking even necessary?

This thesis is outlined as follows: Chapter 2 defines the basic terminology of fuzzing and different types of fuzzers. Chapter 3 presents the ProFuzzBench tool and its usage. Chapter 4 presents our experiments and in Chapter 5 we will discuss the reproducibility of fuzzing. In Chapter 6 we will discuss potential future work and in Chapter 7 we present our conclusion.

Chapter 2

Preliminaries

In this chapter we present background information regarding fuzzing techniques and the underlying technology for ProFuzzBench like docker and AFL. In section 2.1 we talk about the need for fuzzing. In section 2.2, 2.3, we present an overview three high-level fuzzing approaches: blackbox, greybox and white box fuzzing. The feedback metrics like code coverage and bugs are explained in section 2.4. A brief summary of docker and how it works is introduced in section 2.5.

2.1 Fuzzing in general

Fuzzing is a way to detect security vulnerabilities in software. There are many approaches for detecting security vulnerabilities in software.

- **Static program analyzers** that check for code structure like missing a semicolon or missing brackets and are fast in flagging those kinds of mistakes. It can also look if data objects in code are properly used (data flow analysis). However, static program analyzers are prone to false positives and they, unfortunately, won't catch every bug.
- **Code inspection**, this consists of peer-reviewing code, which is already a common practice in software development. Another form of manual code inspection is **penetration testing** where security experts review code where the focus is on the security of the code rather than its functionality of the code. Penetration testing applies to any software and does not require much tooling. However, penetration testing is very labor-intensive and people who do penetration testing are very specialized and they are in high demand[2].

This is where fuzzing can fit in between those two methods.

2.2 White- and blackbox fuzzers

There are many approaches of fuzzing. Most common ones of fuzzing are greybox fuzzing, blackbox fuzzing and whitebox fuzzing.

Blackbox fuzzing often involves using generated input on software where we do not know the internal implementation of the SUT. Blackbox fuzzing requires some given knowledge of the input format[3]. Blackbox fuzzers can make use of a approach which most commonly used in grammar-based fuzzers which is using a given model or grammar. However, the downside of those fuzzers is that it requires some effort in creating those models or grammars. Another approach is supplying the fuzzer with a set of inputs where the a part of the input is mutated in hopes to finding bugs. This approach is commonly used in mutation-based fuzzers.

We have blackbox fuzzers where we do not know the internal implementation of the SUT. With whitebox fuzzzers on the other hand we have full knowledge of the internal implementations of the SUT. This means we have can see and analyse all of the code of the SUT. We can construct interesting inputs to trigger certain branches. Whitebox fuzzers typically use symbolic execution to gather constraints on inputs from branches encountered along the execution[2]. With this the fuzzer can execute a new path with each execution. This may take some time when dealing with larger programs which typically have many paths.

2.3 Greybox fuzzers

We have discussed two ends of the fuzzing spectrum but there is another approach of fuzzing which lies between blackbox and whitebox fuzzing, greybox fuzzing. Greybox fuzzers can observe some aspects of the SUT and use the feedback to discover new paths. Their input is mutated after every execution(like blackbox fuzzers) and use the feedback to make a meaningful mutation for the next execution. One of the most popular greybox fuzzers is AFL which is discussed in section 4.3.

2.4 Feedback metrics

We need some sort of metric to evaluate the performance of a fuzzer on a target (SUT). In this section we explain what code coverage entails and how fuzzers can use these metrics as feedback.

2.4.1 Coverage

One metric that is commonly used to measure the quality of a test is code coverage, where the number of lines executed by a set of tests is divided

by the number of lines of the program. This gives a broad idea of the thoroughness of the test[4]. Code coverage can be used at different levels of the code, in this work we mainly use line coverage and branch coverage.

Branch coverage, this measures the number of execution paths (e.g if-statements, switch cases) that the fuzzer has found during execution. Knowing which paths are executed can give a better insight of the flow of the target. Branch coverage is mainly useful for whitebox and greybox fuzzing due to the required internal knowledge of the target.

Line coverage, this measures the number of statements covered by the fuzzers. The difference with line coverage and branch coverage is that a high line coverage does not automatically mean a high branch coverage. Take this if-statement for example:

```
1    if(cond) {
2        line1();
3        line2();
4        line3();
5        line4();
6    } else {
7        line5();
8    }
```

The line coverage will be 80 percent while the branch coverage is only 50 percent. Nonetheless, line coverage already gives meaningful insights on how good a fuzzers works on a SUT, if a fuzzers does not have good line coverage then the branch coverage will be even worse in a SUT where programs are even larger then shown in the example.

2.4.2 Bugs

During the process of fuzzing the SUT, we see bugs as the number of times the SUT crashes or hangs. Any fuzzers can also measure the number of crashes that happened during the fuzzing of the SUT, a crash is unique if it is not equivalent to previously occurred crashes. These unique crashes could give insight on to why the SUT crashes on this input and where in the code this happened, which means that we have found a bug. This gives us a performance metric for fuzzers with the amount of crashes or hangs found in the SUT. Fuzzers can be tested on how many known bugs it can find in the SUT, but finding unique bugs usually a manual process mainly due to bugs causing errors at several lines and different parts of the SUT. This makes using bugs as a heuristic not ideal during fuzzing but it is great as a performance evaluation after fuzzing.

2.5 Docker

ProFuzzBench uses docker to automate the building, compiling and running of the fuzzers and the targets. Docker is a software platform that enables

rapid building, testing, and deployment of applications. It accomplishes this by packaging software into standardized containers that contain all the necessary components, such as libraries, system tools, code, and runtime, needed for the software to operate. By utilizing Docker, applications can be swiftly deployed and scaled in any environment with the assurance that the code will function properly.

Chapter 3

ProFuzzBench

ProFuzzBench (Protocol Fuzzing Benchmark) is a benchmark for fuzzing of stateful systems that are defined on network protocols like SSH, TLS and FTP [5]. These protocols are packaged in open-source software in order to do actual fuzzing. ProFuzzBench provides a framework for automating, configuration and execution for fuzzing a stateful system. In section 3.1 we discuss how to install ProFuzzBench. Furhtermore, We give an example how of fuzzing a target with ProFuzzBench in section 3.2.

3.1 Workflow

ProFuzzBench uses a set of utilities to automate the compiling, running and analyzing of the target. ProFuzzBench uses docker containers for this automation and this is represented in figure 3.1. During the build process, the fuzzers and the SUT are downloaded from their respective repositories and compiled. This happens during the building of the docker container itself. In the second stage, the fuzzing is done inside the docker container and those results are copied to the host machine. The third stage happens on the host machine where coverage is computed and plotted.

The aim of ProFuzzBench is that we can do this workflow with other fuzzers with a similar amount of steps.

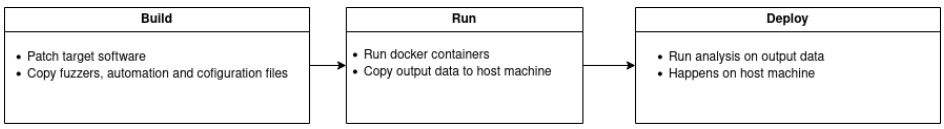

Figure 3.1: ProFuzzBench workflow using docker containers

3.2 Installing ProFuzzBench

The ProFuzzBench repository[1] has quite detailed documentation on how to install ProFuzzBench on a local machine including instructions on building the docker images of each protocol.

3.3 Running ProFuzzBench on LightFTP

If we want to run fuzzing on a protocol (for example FTP) we first need to build the docker image containing the software (in this case lightFTP). This can be done with the following command:

```
cd $PFBENCH
cd subjects/FTP/LightFTP
docker build . -t lightftp
```

We need to store our results, so we make a folder and pass that path to ProFuzzBench, we can then run both AFLnet and AFLnwe for 60 minutes (3600 seconds):

```
mkdir results-lightftp

profuzzbench_exec_common.sh lightftp 4 results-lightftp aflnet
    out-lightftp-aflnet "-P FTP -D 10000 -q 3 -s 3 -E -K -m
    none" 3600 5 &
profuzzbench_exec_common.sh lightftp 4 results-lightftp aflnwe
    out-lightftp-aflnwe "-D 10000 -K -m none" 3600 5
```

3.3.1 Analysing the results

The output of AFL fuzzers is in terms of code coverage over time (see chapter 4) and to make plots we need to reformat it into a csv file:

```
cd $PFBENCH/results-lightftp

profuzzbench_generate_csv.sh lightftp 4 aflnet results.csv 0
profuzzbench_generate_csv.sh lightftp 4 aflnwe results.csv 1
```

The `results.csv` file shown in table **??** has six columns showing the timestamp, subject program, fuzzer name, run index, coverage type and its value. The file contains both line coverage and branch coverage over time information. Each coverage type comes with two values, in percentage (_per) and in absolute number (_abs). With this we can make our line coverage and edge coverage plots.

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

time	subject	fuzzer	run	cov_type	cov
1684134470	lightftp	aflnet	1	l_per	37
1684134470	lightftp	aflnet	1	l_abs	417
1684134470	lightftp	aflnet	1	b_per	20.3
1684134470	lightftp	aflnet	1	b_abs	164
1684134471	lightftp	aflnet	1	l_per	38.9
1684134471	lightftp	aflnet	1	l_abs	438
1684134471	lightftp	aflnet	1	b_per	22.5
1684134471	lightftp	aflnet	1	b_abs	182
1684134476	lightftp	aflnet	1	l_per	39.3
1684134476	lightftp	aflnet	1	l_abs	443
1684134476	lightftp	aflnet	1	b_per	23.6
1684134476	lightftp	aflnet	1	b_abs	191
1684134480	lightftp	aflnet	1	l_per	40
1684134480	lightftp	aflnet	1	l_abs	451
1684134480	lightftp	aflnet	1	b_per	24.5
1684134480	lightftp	aflnet	1	b_abs	198
1684134491	lightftp	aflnet	1	l_per	40
1684134491	lightftp	aflnet	1	l_abs	451
1684134491	lightftp	aflnet	1	b_per	24.5
1684134491	lightftp	aflnet	1	b_abs	198
1684134505	lightftp	aflnet	1	l_per	40.9
1684134505	lightftp	aflnet	1	l_abs	461

Table 3.1: Example of the results.csv

Now that we have the results in a usable format and we run the following command to generate the plots:

```
profuzzbench_plot.py -i results.csv -p lightftp -r 4 -c 60 -s 1
    -o cov_over_time.png
```

3.4 Issues

3.4.1 Address sanitizer

However, running ProFuzzBench did not go as smoothly as installing it. Apparently, afl-fuzzers are included in the benchmark crash because afl-fuzzers automatically set a memory limit when they run and that makes the fuzzers crash with the adress sanitizer(a memory error detector for C/C++).

This can be fixed by running the fuzzers with the option `-m none` which removes the memory limit on the fuzzers. This was not an easy thing to debug and there is no documentation on it in the ProFuzzBench repository.

Chapter 4

Extending ProFuzzBench

In this chapter, we present the methodology used to implement new fuzzers to ProFuzzBench and the limitations of ProFuzzBench. Furthermore, we do a case study where we compare SNPSFuzzer[1], AFLnet (which is included by ProFuzzBench), BooFuzz[2] and Nyx-Net on LightFTP[3], an open source FTP server. In section 4.1, we discuss about compatible fuzzers for the experiments. In section 4.2 we present our testing environment. In this testing environment, we extend ProFuzzBench:

- We describe the functionality of AFLnet and AFLnwe. We show how AFLnet and AFLnwe can be build and run on LightFTP in section 4.3.
- We describe the functionality of SNPSFuzzer and discuss the configuration for running SNPSFuzzer on LightFTP in section 4.4. We discuss the results of SNPSFuzzer, AFLnet and AFLnwe in section 4.6.
- We present the modifications done to the analysis script such that we can plot the results from SNPSFuzzer on LightFTP in section 4.5.
- We describe the functionality of BooFuzz and show how we can build and run BooFuzz on LightFTP in section 4.7. We discuss the results of BooFuzz in section 4.8.
- We describe the functionality and configuration of Nyx-Net. We also discuss the results of Nyx-Net on LightFTP in section 4.9

4.1 Finding compatible fuzzers

The ideal of ProFuzzBench is that researcher could use any fuzzer of their liking and extend ProFuzzBench with that fuzzers. However, due to the

[1] https://github.com/SNPSFuzzer/SNPSFuzzer
[2] https://github.com/jtpereyda/boofuzz
[3] https://github.com/hfiref0x/LightFTP

scope of the thesis we only looked at AFL-based fuzzers such that meaningful results could be presented. This is because ProFuzzBench already has an implementation of AFLnet which should make it easier to extend new AFL-based fuzzers with. Daniele et al. provide a comprehensive overview of stateful fuzzers used today[3]. However, while some of these AFL-based fuzzers that were mentioned in the article did not have any source code, making them impractical for this experiment because there is no possible way to run those fuzzers. FFuzz[4] did provide source code but lacked proper documentation of the software dependencies. In table 4.1 we present the fuzzers that were mentioned in the article and tested if they were compatible with ProFuzzBench.

Fuzzer	Based on	Source code	Issues
Nyx-Net	AFL	yes	Missing KVM modules
FFuzz	AFL	yes	No documentation
FitM	AFL	yes	No clear instruction for fuzzing targets
SNPS-fuzzer	AFL	yes	No proper documentation on usage
EPF	AFL	yes	
GANFuzz	seq-gan model	no	
SGPFuzzer	AFL	no	
SPFuzz	AFL	no	
Chen et al	AFL	no	
SeqFuzzer	seq2seq model	no	

Table 4.1: List of fuzzers which were tested if they could be extended with ProFuzzBench

Some of the fuzzers like SGPFuzzer and FFuzz that had source code were discovered solely through searching via GitHub on the repository name, a simple Google search was not enough because of the non-SEO friendly naming of those fuzzers. FitM (Fuzzer In The Middle) [5] did have source code and had proper documentation. However it was not a suitable candidate due to the input requirements of FitM which we could not retrieve from LightFTP due to time limitations. EPF was pretty well documented and did provide source code, but was left out, again due to time limitations. This leaves us with SNPSFuzzer and Nyx-net which we discuss in section 4.4 and section 4.6.

[4]https://github.com/Epeius/FFuzz/tree/master
[5]https://github.com/fgsect/FitM

4.2 Testing Environment

The fuzzing was done on one machine with the following specifications:

Hardware	
OS	Ubuntu 22.04 LTS
CPU	AMD Ryzen 9 5900HX (8 cores, 16 threads)
RAM	32 Gigabytes

Table 4.2: Hardware specifications

The goal of our experiment is to find see how easy it is to fuzz a target (LightFTP) from ProFuzzBench with a fuzzer of chosing. During this experiment we fuzz LightFTP for one hour with AFLnet, AFLnwe and SNPSFuzzer using 4 parallel docker builds. Furthermore, we give detailed explanation on the modifications to integrate SNPSFuzzer to ProFuzzBench.

4.3 AFLnet and AFLnwe

We first look at the results of the fuzzers that come packaged with ProFuzzBench. AFLnet and AFLnwe are both based on AFL. However, AFLnwe differs in that it is just a "network-enabled" version of AFL where AFLnwe sends mutated inputs over TCP/IP socket instead of using a file I/O[6]. AFLnet is much more modified fork of AFL. AFLnet organizes an input as session with multiple messages and can mutate messages at the message level[6][7]. This means it can drop and duplicate messages rather than mutating bytes which is done with AFLnwe.

4.3.1 Building and running AFLnet and AFLnwe

ProFuzzBench already provides a dockerfile where we can build AFLnet and AFLnwe together with LightFTP. We only need to run this command inside of the ProFuzzBench repository:

```
cd $PFBENCH
cd subjects/FTP/LightFTP
docker build . -t lightftp-afl
```

We fuzz LightFTP for 1 hour with AFLnet and AFLnwe using this command:

```
profuzzbench_exec_common.sh lightftp 4 results-lightftp aflnet
    out-lightftp-aflnet "-P FTP -D 10000 -q 3 -s 3 -E -K -m
    none" 3600 5 &
profuzzbench_exec_common.sh lightftp 4 results-lightftp aflnwe
    out-lightftp-aflnwe "-D 10000 -K -m none" 3600 5
```

We can use specific run time parameters of AFLnet to control the fuzzing process:

1. (`-P`) is used to specify which protocol is used, in this case FTP.
2. (`-D`) is the optional waiting time for the server to complete initialization before AFLnet starts fuzzing.
3. State selection (`-q`) and seed selection (`-s`) algorithm are both set to `favor`.
4. We use (`-K`) option that allows the SUT to shutdown gracefully.
5. We enable state-aware mode using the (`-E`).
6. We disable the memory-limit imposed by ASAN by using the (`-m none`) option. Both AFLnwe and AFLnet fail to start fuzzing without this option.

4.4 SNPSFuzzer

SNPSFuzzer is a greybox fuzzer for stateful systems using snapshots. SNPSFuzzer restores the SUT to a specific state that the fuzzer can do fuzzing on. This should improve the message processing speed and increase edge coverage[8]. SNPSFuzzer also makes us of a message chain algorithm which in mutates messages in a certain message chain in order to get to a new state in the SUT. We need to package SNPSFuzzer together with the LightFTP software in a Dockerfile. With this Dockerfile, we can make a docker image which enables us to run the experiment with the same environment and correct environment variables every time. ProFuzzBench recommends to build the new fuzzer on a new docker image seperate from the one that already contains AFLnet and AFLnwe. This is because AFLnet sets environment variables which SNPSFuzzer recognizes due to being based on AFL. Furthermore, this also creates issues with the `afl-fuzz` compiler which both AFLnet and SNPSFuzzer use.

4.4.1 Building and running SNPSFuzzer

SNPSFuzzer uses the `python-protobuf` package for compiling the fuzzer itself. `python-protobuf` is a python2 package and python2 is no longer supported and most modern linux distributions already transitioned to python3. Nonetheless, using docker we can build an old Ubuntu 16.04 (or 18.04) and install `python-protobuf` directly with no issues. In appendix A.1 is a complete implementation of SNPSFuzzer with docker. We also need to modify the provided `run.sh` file which the docker containers use such that we can run SNPSFuzzer with the `profuzzbench_exec_common.sh` such that it follows the ProFuzzBench workflow (see section 3.1). The `run.sh` starts with this:

```
#!/bin/bash
#Commands for afl-based fuzzers (e.g., aflnet, aflnwe)
if $(strstr $FUZZER "afl"); then
    # Fuzzing
```

It checks if the fuzzer argument which is given when `profuzzbench_exec_common.sh` is run, contains the word 'afl' which is not in SNPSFuzzer. We modify `run.sh` as follows:

```
#!/bin/bash
#Commands for afl-based fuzzers (e.g., aflnet, aflnwe)
if [[ $FUZZER = "aflnet" ]] || [[ $FUZZER = "aflnwe" ]] || [[
    $FUZZER = "SNPSFuzzer" ]]; then
    #Fuzzing
```

The `run.sh` runs AFLnet and AFLnwe as follows:

```
#!/bin/bash
#Commands for afl-based fuzzers (e.g., aflnet, aflnwe)
if $(strstr $FUZZER "afl"); then

  # Run fuzzer-specific commands (if any)
  if [ -e ${WORKDIR}/run-${FUZZER} ]; then
    source ${WORKDIR}/run-${FUZZER}
  fi

  TARGET_DIR=${TARGET_DIR:-"LightFTP"}
  INPUTS=${INPUTS:-"${WORKDIR}/in-ftp"}

  #Step-1. Do Fuzzing
  #Move to fuzzing folder
  cd $WORKDIR/${TARGET_DIR}/Source/Release
  echo "$WORKDIR/${TARGET_DIR}/Source/Release"
  timeout -k 0 --preserve-status $TIMEOUT /home/ubuntu/${FUZZER
    }/afl-fuzz -d -i ${INPUTS} -x ${WORKDIR}/ftp.dict -o
    $OUTDIR -N tcp://127.0.0.1/2200 $OPTIONS -c ${WORKDIR}/
    ftpclean ./fftp fftp.conf 2200
```

Due to SNPSFuzzer being based on AFL and thus uses the same[6] `afl-fuzz` compiler on line 29. We only need to modify a bit to make the script run SNPSFuzzer:

```
#!/bin/bash
#Commands for afl-based fuzzers (e.g., aflnet, aflnwe)
if [[ $FUZZER = "aflnet" ]] || [[ $FUZZER = "aflnwe" ]] || [[
    $FUZZER = "SNPSFuzzer" ]]; then

  # Run fuzzer-specific commands (if any)
  if [ -e ${WORKDIR}/run-${FUZZER} ]; then
    source ${WORKDIR}/run-${FUZZER}
  fi

  TARGET_DIR=${TARGET_DIR:-"LightFTP"}
```

[6]Same in name and structure but not in configuration and environment variables!

```
  INPUTS=${INPUTS:-"${WORKDIR}/in-ftp"}

  #Step-1. Do Fuzzing
  #Move to fuzzing folder
  cd $WORKDIR/${TARGET_DIR}/Source/Release
  echo "$WORKDIR/${TARGET_DIR}/Source/Release"
  if [ $FUZZER == "SNPSFuzzer" ]; then
    timeout -k 0 --preserve-status $TIMEOUT /home/ubuntu/${
    FUZZER}/afl-fuzz -d -i /home/ubuntu/SNPSFuzzer/tutorials/
    lightftp/in-ftp -x /home/ubuntu/SNPSFuzzer/tutorials/
    lightftp/ftp.dict -o $OUTDIR -N tcp://127.0.0.1/2200
    $OPTIONS -c ${WORKDIR}/ftpclean  ./fftp fftp.conf 2200
  else
    timeout -k 0 --preserve-status $TIMEOUT /home/ubuntu/${
    FUZZER}/afl-fuzz -d -i ${INPUTS} -x ${WORKDIR}/ftp.dict -o
    $OUTDIR -N tcp://127.0.0.1/2200 $OPTIONS -c ${WORKDIR}/
    ftpclean ./fftp fftp.conf 2200
  fi
  STATUS=$?
```

Now we can run SNPSFuzzer on each docker container, see appendix A.2 for the full code of `run.sh`.

4.5 ProFuzzBench plotting modifications

The third step in the ProFuzzBench workflow(see figure 3.1) is analysis. Here ProFuzzBench already provides a script to get line and edge coverage of the output in a CSV format. However, we again need to make some modifications to make it also plot the line and edge coverages from SNPSFuzzer. Fortunately, it is very simple fix that we need to do in `profuzzbench_plot.py`

```
def main(csv_file, put, runs, cut_off, step, out_file):
  #Read the results
  df = read_csv(csv_file)

  mean_list = []

  for subject in [put]:
    for fuzzer in ['aflnet', 'aflnwe']:
```

Must be changed to:

```
def main(csv_file, put, runs, cut_off, step, out_file):
  #Read the results
  df = read_csv(csv_file)

  #Calculate the mean of code coverage
  #Store in a list first for efficiency
  mean_list = []
  fuzzers = df.fuzzer.unique()

  for subject in [put]:
    for fuzzer in fuzzers:
```

There is no need to hardcode the names of the fuzzers. By avoiding this, ProFuzzBench becomes more configurable, saving time on bug fixing.

4.6 Coverage results of AFLnet, AFLnwe and SNPSFuzzer

	SNPSFuzzer (no mca)	SNPSFuzzer	AFLnet	AFLnwe
Executions per second	7,5725	6,97	7,96	40,9125
Total executions	28860	29809	33291	147746
Total paths	274	264	279	69
Paths found	272	262	277	67

Table 4.3: Additional fuzzing results of LightFTP using SNPSFuzzer, AFLnet and AFLnwe

From figure 4.1, 4.2 and table 4.3, we notice that SNPSFuzzer performs almost identical or sometimes worse then AFLnet. We find this strange because one would assume that SNPSFuzzer which is based on AFLnet should perform better when it also uses snapshotting. [8] claimes that SNPSFuzzer should have find more paths in LightFTP than AFLnet. We suspect that the snapshotting does not function correctly with LightFTP and we were unable to find a workaround due to poor documentation.

We see that AFLnwe has the highest throughput in terms of executions per second. However, AFLnwe is not a message-oriented fuzzer (like AFLnet and SNPSFuzzer), which sends messages and analyzes the receiving message, as AFLnwe just sends a uninterrupted stream of bytes. This explains the high amount of executions per second. This also explains why AFLnwe does not find much paths in comparison to AFLnet and SNPSFuzzer.

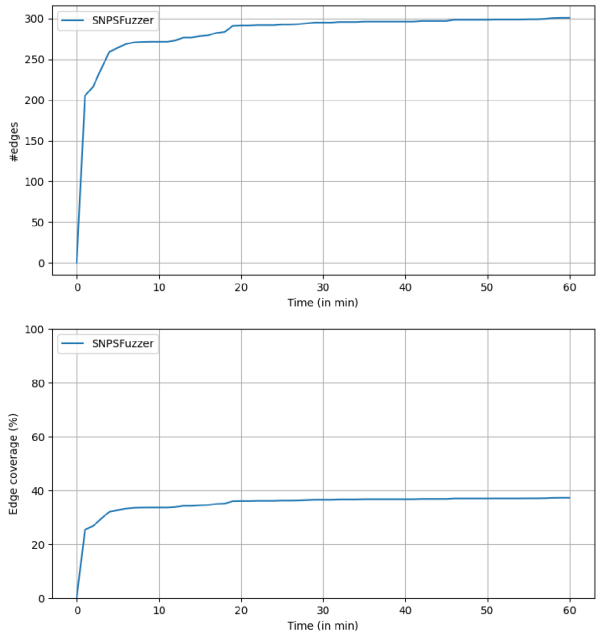

Figure 4.1: Edge coverage SNPSFuzzer on LightFTP

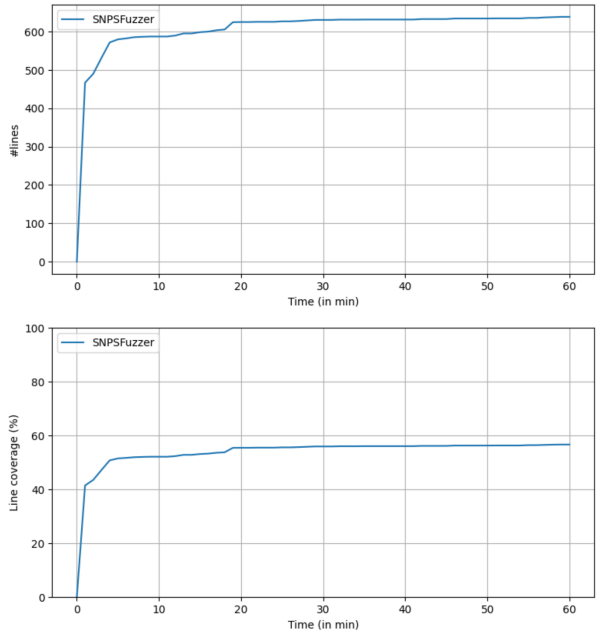

Figure 4.2: Line coverage SNPSFuzzer on LightFTP

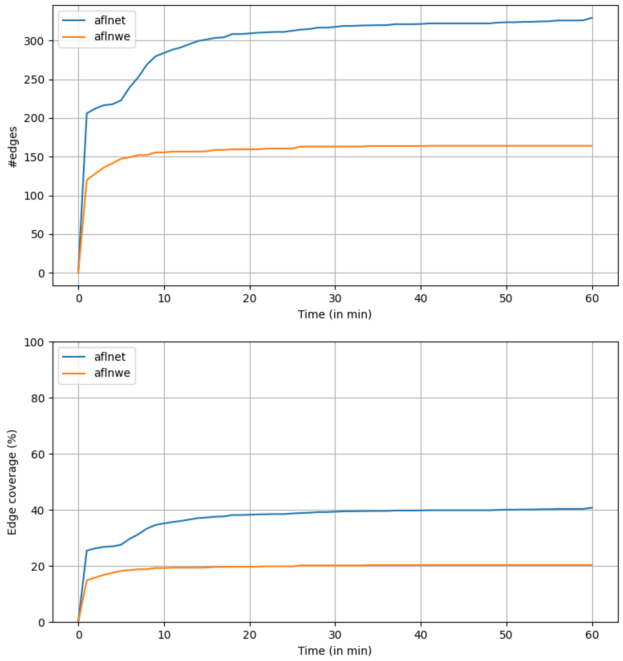

Figure 4.3: Edge coverage of AFLnet and AFLnwe on LightFTP

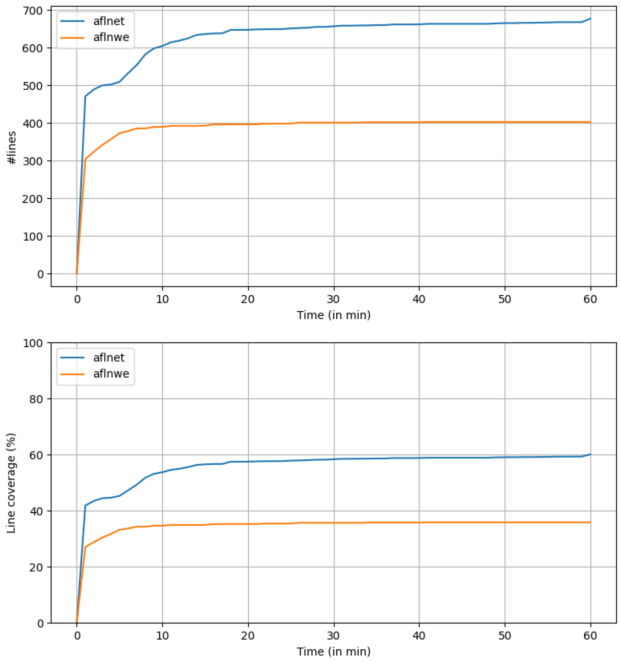

Figure 4.4: Line coverage of AFLnet and AFLnwe on LightFTP

4.7 BooFuzz

We also wanted to look at the performance of a blackbox fuzzer on LightFTP. For this thesis, we used BooFuzz, a grammar based fuzzer. BooFuzz uses a provided grammar and mutates it to fuzz specific target that accept that grammar. Furthermore, when the SUT crashes, BooFuzz logs the mutated output which caused the crash and restarts the SUT for further fuzzing. For this experiment we use a simple FTP grammar (see appendix A.3) and fuzz LightFTP for one hour. Because BooFuzz does not use coverage and has no knowledge of the iternal implementation of LightFTP, BooFuzz can not be benchmarked with ProFuzzBench.

4.7.1 Building and running BooFuzz

Installing BooFuzz is straightforward with the `pip install boofuzz` command and BooFuzz is in comparison with the fuzzers we discussed very documentated. The SUT (LightFTP) needs to be already running before we begin fuzzing with BooFuzz. We can manually run the LightFTP (version 2.0 for fuzzing) server by compiling LightFTP using the AFL compiler:

```
1     cd /path/to/LightFTP/Source/Release #go to executable
    LightFTP
```

```
    CC=/path/to/AFL/AFL-clang-fast make clean all #compile
    LightFTP
    ./fftp fftp.conf 2200 #run LightFTP on port 2200
```

With `fftp.conf` we configure the server such that it accepts certain users like 'ubuntu' that have a specific password and run it on localhost with port 2200. Now BooFuzz can fuzz on the LightFTP server using that port. We then need to run the following command to run BooFuzz for 1 hour:

```
    time python3 boofuzz_lightftp.py 3600
```

4.8 Performance results of BooFuzz on LightFTP

Because BooFuzz only can register crashes we can only measure how many crashes it finds. BooFuzz performed in total 135287 executions resulting in a executions speed of 37,6 executions per second. However, after one hour there were no crashes found with BooFuzz.

4.9 Nyx-Net

We also tried Nyx-Net[7], a snapshot-based fuzzer. This fuzzer was most interesting because the researchers used ProFuzzBench in their paper to fuzz their targets. Nyx-Net uses a hypervisor for their snapshots[9]. Hypervisors can run between the hardware and the guest-OS (type-1 hypervisor/bare-metal hypervisors) or one can run the hypervisor between the host-OS and the guest-OS[10]. To be more specific, Nyx-Net uses a modified version of QEMU[8] and KVM[9] that allows the Linux kernel to function as a hypervisor[11]. KVM uses hardware virtualization extensions provided by modern CPU to run the guest-OS inside of the VM natively. This setup ensures high performance virtualization[9].

4.9.1 Building and running Nyx-Net

Because we can not use docker with the modified version of QEMU that Nyx-Net uses, we choose to use a Ubuntu 21.04 VM to install and use Nyx-Net. The reason that we use Ubuntu 21.04 is to ensure we have similar setup as [9] suggest. We already run into another issue that Ubuntu 21.04 is no longer supported and using the standard `sudo apt update && apt upgrade` does not work out of the box, but this is easily mitigated by using an archive of the ubuntu package sources. This can be done using the following command:

[7] https://github.com/RUB-SysSec/nyx-net
[8] Quick Emulator
[9] Kernel-based Virtual Machine

```
sudo sed -i -re 's/([a-z]{2}\.)?archive.ubuntu.com|security.ubuntu.com/old-releases.ubuntu.com/g' /etc/apt/sources.list
```

We can then compile Nyx-Net using the `setup.sh` script in the Nyx-Net repository:

```
cd nyx-net
./setup.sh
```

Another issue with the current codebase (which also uses Rust) of Nyx-Net is that it is not compatible with the current version of Rust[10]. The rust compilor gives the following error message:

```
error: reference to packed field is unaligned
|
222 | &self.feedback_data.shared.ijon.max_data
|
= warning: this was previously accepted by the compiler but is being phased out; it will become a hard error in a future release!
```

We avoid this error by using a older rust version[11]. Now that we have the fuzzer compiled, we need to build LightFTP to make it compatible with Nyx-Net. Fortunately, Nyx-Net already provides a script which builds and packages LightFTP in a format which Nyx-Net can do fuzzing on. We run this script using the following command:

```
./nyx-net/targets/packer_scripts/pack_lightFTP.sh
```

Before we can fuzz LightFTP with Nyx-Net we need to enable the KVM module on our virtual machine. We use the following commands to enable the KVM:

```
sudo modprobe -r kvm_amd  #unload kvm module specific for AMD CPU's
sudo modprobe -r kvm      #unload standard kvm module
sudo modprobe kvm enable_vmware_backdoor=y  #reload standard kvm module with vmware backdoor enabled
sudo modprobe kvm_amd     #reload kvm module specific for AMD CPU's
```

To run Nyx-Net on LightFTP we need to run the following command:

```
cargo run --release -- -s ../../targets/packed_targets/nyx_lightftp/
```

We let Nyx-Net fuzz LightFTP for one hour but unfortunately the results were quite disappointing. From figure 4.4 we notice that Nyx-Net hangs at around 26,5 percent line coverage and 15,4 percent edge coverage. Nyx-net should perform 10 percent better than AFLnet [9].

[10] version 1.69.0
[11] version 1.56

Time	Fuzzer	Lines_cov	Lines_abs	Edge_cov	Edge_abs
1685898808	nyx-aggressive	25.3	286	13,4	108
1685898810	nyx-aggressive	25.3	286	13,4	108
1685898812	nyx-aggressive	25.3	286	13,4	108
1685898813	nyx-aggressive	25.3	286	13,4	108
1685898814	nyx-aggressive	25.3	286	13,4	108
1685898816	nyx-aggressive	25.3	286	13,4	108
1685898818	nyx-aggressive	25.3	286	13,4	108
1685898822	nyx-aggressive	25.3	286	13,4	108
1685898828	nyx-aggressive	26.3	300	15,4	124
1685898828	nyx-aggressive	26.3	300	15,4	124

Table 4.4: Coverage data points of Nyx-Net on LightFTP

We reached out to one of the researchers involved in the development of Nyx-Net, and they informed us that the public version of the Nyx-Net codebase, which was outdated, was not utilized for the paper. We noticed similarities with the data[12] they collected using the outdated version of Nyx-Net with our results. The updated version of Nyx-Net[13] did match with the results from [9]. However, we can not use those results because we have no way to verify those results.

[12] `https://github.com/RUB-SysSec/nyx-net-profuzzbench/blob/ab5078b3eda2f433eaac5fd3d6c640f7ffc85e72/data/noseeds/lightftp.csv`

[13] `https://github.com/RUB-SysSec/nyx-net-profuzzbench/blob/ab5078b3eda2f433eaac5fd3d6c640f7ffc85e72/data/with-newspec/lightftp.csv`

Chapter 5

Reproducibility of fuzzing

As we can see from the results of SNPSFuzzer and Nyx-Net, we were not able to reproduce the results which the researchers of those fuzzers claimed in their respective papers[8, 9]. Even though the hardware differs from those papers there should be at least a minimal improvement compared to AFLnet. We also saw that some fuzzers discussed in this thesis(see section 4.1) did not provide source code or proper documentation of that source code.

We are deeply concerned about reproducibility of the scientific results from those fuzzers. One should assume that a scientific paper where a researcher shows that for example fuzzer A performs significantly better than fuzzer B that those results can be reproduced by different researcher using a different machine.

However, this what not the case with the experiments reported in section 4.6 of this thesis. One cause maybe that the source code is not maintained after the publication of the research. This was the case with Nyx-Net where the researcher told us that the source code of Nyx-Net, which was public, was not updated after the paper[9] was published.

Figure 5.1: ACM artifacts badges [12]

5.1 ACM artifact review for reproducibility

To further improve the repoducibility of fuzzers in research, ACM has partially adopted the process of artifact review[12]. An artifact in this case is a digital object that was created and used by the authors in experiments. This could be whole software systems, scripts to run the experiments, input data or raw data collected in the experiment. This artifacts are also submitted and reviewed. The corresponding paper will have badge that will state whether artifacts were available, functional or reusable and if the results were even reproducible (see figure 5.1).

Chapter 6

Future Work

The subsequent sections provide a summary of research recommendations aimed at extending ProFuzzBench even further.

6.1 Adding more fuzzers to ProFuzzBench

We only looked at adding three fuzzer to ProFuzzBench, extending ProFuzzBench even further would offer more interesting comparisons with SNPSFuzzer and AFLnet. For example, we can look at fuzzers like FitM[13] which needs some further investigation how its inputs work in order for FitM to work on ProFuzzBench. This could provide yet another meaningful comparison to AFLnet which is also based on AFL. Daniele et al. also provide a lis of other fuzzers which were not based on AFL like for example AspFuzz[14]. A nice inclusion to that article would be to also specify which fuzzers do have or did not provide source code.

6.2 Extending the duration of fuzzing

Another area for further research could be the duration of the fuzzing. The duration could be extended to for example like 24 hours as 1 hour of fuzzing LightFTP did not result in any crashes. However, we have doubts if fuzzing for longer would improve the code coverage of SNPSFuzzer or AFLnet. From figure 4.2 and 4.4 we notice no significant improvement in code coverage after 30 minutes.

6.3 Testing on different SUT's

Furthermore, this thesis only looked at one target, LightFTP. Using more targets may give a more comprehensive analysis of fuzzers on multiple targets. One could look at the performance of SNPSFuzzer and AFLnet on for example Live555 (a RTSP server) or OpenSSH.

6.4 Adding coverage to blackbox fuzzers

There is also further research to be done on getting code coverage from blackbox fuzzers. BooFuzz does have an open pull request where it adds code coverage to BooFuzz using AFL instrumentation[1]. This pull request is not merged yet and thus not in the latest version of BooFuzz.

6.5 Vulnerabilities as a performance metric

We could use also look at vulnerabilities from CVEs or from OSS-Fuzz and use those vulnerabilities as a metric whether a fuzzer can find them or not. This approach is also used for AFLnet in [7] where AFLnet, AFLnwe and BooFuzz are compared on how fast the fuzzers can find known CVEs.

OSS-Fuzz[2] is an open-source project maintained by Google which provides a framework for open source software to implement fuzzing technique for finding more security vulnerabilities. It also provides a platform[3] where users can report bugs they found through fuzzing, such it can be fixed quickly. One could use this platform to check for vulnerabilities in one of the targets of ProFuzzBench. However, OSS-fuzz only supports stateless fuzzing to find vulnerabilities in stateful SUTs, so it can be the case that a stateful fuzzer could find more bugs than what is reported in OSS-fuzz.

[1] https://github.com/jtpereyda/boofuzz/pull/508
[2] https://google.github.io/oss-fuzz/
[3] https://bugs.chromium.org/p/oss-fuzz/issues/list

Chapter 7

Conclusions

At first glance, fuzzing appears to be a simple and automated method to discover bugs and vulnerabilities. However, a significant amount of underlying groundwork is required to ensure fuzzers work as intended. Think of complications like software dependencies or incorrect compiler versions, which could hours to debug. Consider, too, whether the fuzzer can operate within a Docker container. For instance, it took approximately two weeks to get SNPSFuzzer to compile and run inside of a Docker container. Furthermore, it took us a month[1] to successfully set up Nyx-Net, but even then, it failed to generate significant results due to outdated source code.

We then needed to figure out if we can run SNPSFuzzer and Nyx-Net on LightFTP. It takes a considerable amount of time to make a specific SUT like LightFTP work with SNPSFuzzer and Nyx-Net.

Nevertheless, once we successfully executed SNPSFuzzer and Nyx-Net on LightFTP, Further experimentation, like longer duration, on LightFTP with SNPSFuzzer became relatively straightforward. In the case of SNPSFuzzer, we accomplished this by building a Docker container in which we built and compiled both SNPSFuzzer and LightFTP, allowing us to subsequently run SNPSFuzzer on LightFTP. Additionally, for Nyx-Net, we employed a virtual machine configuration where Nyx-Net ran on a LightFTP server. To ensure that we ran Nyx-Net on the same LightFTP server, we build a docker container which only contained LightFTP. This container ran inside of the virtual machine.

Regrettably, should we wish to utilize a different SUT, it becomes necessary once more to modify the SUT to ensure that we can run the SUT with our fuzzers, and proceed with creating a docker container for the new SUT following the ProFuzzBench workflow (see figure 3.1). This needs to be repeated for each new fuzzer.

We can also look at blackbox fuzzers like BooFuzz are suited for fuzzing a SUT like LightFTP but it can not perform well without building a proper

[1] working full time 2 days per week

grammar which must be done for every SUT.

Furthermore, it is hard to compare blackbox fuzzers like BooFuzz with greybox fuzzers like AFL. This is because BooFuzz does not use coverage to mutate it messages and does not output coverage which makes it difficult to compare it with AFL fuzzers like AFLnet and SNPSFuzzer. The only metrics we could use to compare AFL fuzzers with BooFuzz is the number of crashes and the executions per second. However, there are some advancements in getting code coverage from blackbox fuzzers like BooFuzz (see section 6.4)

We also discussed about the reproducibility of fuzzing. We noticed that many researchers do not provide source code for their fuzzers (see table 4.1) and thus providing no way to replicate their results. We do see that publishers such as ACM are trying to encourage researchers to make their research more reproducible using artifact review (see chapter 5). From our experiments we concluded that it is not easy to extend ProFuzzBench with more fuzzers, but do see the potential on how it can make artifact review easier and thus improving reproducibility.

Bibliography

[1] 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.

[2] Patrice Godefroid. Fuzzing: hack, art, and science. *Communications of the ACM*, 63(2):70–76, 2020.

[3] Cristian Daniele, Seyed Behnam Andarzian, and Erik Poll. Fuzzers for stateful systems: Survey and research directions. `https://arxiv.org/abs/2301.02490`, 2023.

[4] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 560–564, 2015.

[5] Roberto Natella and Van-Thuan Pham. ProFuzzBench: a benchmark for stateful protocol fuzzing. *ACM, International Symposium on Software Testing and Analysis*, 2021.

[6] Roberto Natella. StateAFL: Greybox fuzzing for stateful network servers. *Springer, Empirical Software Engineering*, 27(7):191, 2022.

[7] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLnet: A Greybox Fuzzer for Network Protocols. In *13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE, 2020.

[8] Junqiang Li, Senyi Li, Gang Sun, Ting Chen, and Hongfang Yu. SNPS-Fuzzer: A Fast Greybox Fuzzer for Stateful Network Protocols Using Snapshots. *IEEE Transactions on Information Forensics and Security*, 17:2673–2687, 2022.

[9] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-Net: Network Fuzzing with Incremental Snapshots. In *Proceedings of the Seventeenth European Conference on Computer Systems*. ACM, 2022.

[10] Robert P Goldberg. Architectural principles for virtual computer systems. Technical report, Harvard University, 1973.

[11] Gernot Heiser. The role of virtualization in embedded systems. In *1st Workshop on Isolation and Integration in Embedded Systems*, pages 11–16. ACM, Apr 2008.

[12] Artifact review and badging - current. `https://www.acm.org/publications/policies/artifact-review-and-badging-current`, Aug 2020.

[13] Dominik Maier, Otto Bittner, Marc Munier, and Julian Beier. FitM: Binary-Only Coverage-Guided Fuzzing for Stateful Network Protocols. In *Workshop on Binary Analysis Research (BAR)*. Internet Society, 2022.

[14] Takahisa Kitagawa, Miyuki Hanaoka, and Kenji Kono. AspFuzz: A state-aware protocol fuzzer based on application-layer protocols. In *The IEEE symposium on Computers and Communications*, pages 202–208, 2010.

[15] Yingchao Yu, Zuoning Chen, Shuitao Gan, and X. F. Wang. SGP-Fuzzer: A State-Driven Smart Graybox Protocol Fuzzer for Network Protocol Implementations. *IEEE Access*, 8:198668–198678, 2020.

[16] Wu Biao, Tang Chaojing, and Zhang Bin. FFUZZ: A Fast Fuzzing Test Method for Stateful Network Protocol Implementation. In *2021 2nd International Conference on Computer Communication and Network Security (CCNS)*, pages 75–79. IEEE, 2021.

Appendix A

Appendix

A.1 Dockerfile SNPSFuzzer

```
FROM ubuntu:16.04

# Install common dependencies
ENV DEBIAN_FRONTEND=noninteractive
RUN apt-get -y update && \
    apt-get -y install sudo \
    apt-utils \
    build-essential \
    openssl \
    clang \
    graphviz-dev \
    git \
    autoconf \
    libgnutls28-dev \
    libssl-dev \
    llvm \
    python3-pip \
    nano \
    net-tools \
    vim \
    gdb \
    netcat \
    strace \
    wget

# Set up fuzzers
RUN sudo apt-get -y install libprotobuf-dev \
    libprotobuf-c-dev \
    protobuf-c-compiler\
    protobuf-compiler \
    python-protobuf \
    libnl-3-dev \
    libnet-dev \
    libbsd-dev \
    libaio-dev \
```

```
36      libcap-dev \
37      pkg-config
38
39  RUN sudo apt-get -y --no-install-recommends install asciidoc
40  RUN sudo apt-get -y --no-install-recommends install xmlto
41
42  # Add a new user ubuntu, pass: ubuntu
43  RUN groupadd ubuntu && \
44      useradd -rm -d /home/ubuntu -s /bin/bash -g ubuntu -G sudo
        -u 1000 ubuntu -p "$(openssl passwd -1 ubuntu)"
45
46  RUN chmod 777 /tmp
47
48  RUN echo '%sudo ALL=(ALL) NOPASSWD:ALL' >> \
49  /etc/sudoers
50
51  RUN pip3 install gcovr==4.2
52
53  # Use ubuntu as default username
54  USER ubuntu
55  WORKDIR /home/ubuntu
56
57  # Import environment variable to pass as parameter to make (e.g
        ., to make parallel builds with -j)
58  ARG MAKE_OPT
59
60  # Set up fuzzers
61  RUN git clone https://github.com/SNPSFuzzer/SNPSFuzzer.git && \
62      cd SNPSFuzzer && \
63      cd criu4snpsfuzzer && \
64      make  && sudo make install && \
65      sudo cp ./lib/c/libcriu.so /usr/local/lib/libcriu.so.2 && \
66      sudo ldconfig && \
67      cd ~
68
69  ENV SNPSFUZZER="/home/ubuntu/SNPSFuzzer"
70
71  RUN cd ~
72  RUN cd $SNPSFUZZER && make &&\
73      cd llvm_mode && make &&\
74      cd .. &&\
75      sudo make install
76
77  # Set up environment variables for SNPSFUZZER
78  ENV WORKDIR="/home/ubuntu/experiments"
79  ENV LLVM_CONFIG="llvm-config-3.8"
80  ENV PATH="${PATH}:${SNPSFUZZER}:/home/ubuntu/.local/bin:${
        WORKDIR}"
81  ENV AFL_PATH="${SNPSFUZZER}"
82  ENV AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1 \
83      AFL_SKIP_CPUFREQ=1 \
84      AFL_NO_AFFINITY=1
85
86  RUN mkdir $WORKDIR
```

```
87
88 COPY --chown=ubuntu:ubuntu fuzzing.patch ${WORKDIR}/fuzzing.
       patch
89 COPY --chown=ubuntu:ubuntu gcov.patch ${WORKDIR}/gcov.patch
90
91 # Set up environment variables for ASAN
92 ENV ASAN_OPTIONS='abort_on_error=1:symbolize=0:detect_leaks=0:
       detect_stack_use_after_return=1:detect_container_overflow
       =0:poison_array_cookie=0:malloc_fill_byte=0:
       max_malloc_fill_size=16777216'
93
94 # Download and compile LightFTP for fuzzing
95 RUN cd $WORKDIR && \
96     git clone https://github.com/hfiref0x/LightFTP.git && \
97     cd LightFTP && \
98     git checkout 5980ea1 && \
99     patch -p1 < ${WORKDIR}/fuzzing.patch && \
100    cd Source/Release && \
101    AFL_USE_ASAN=1 CC=afl-clang-fast make clean all $MAKE_OPT
102
103 # Set up LightFTP for fuzzing
104 RUN cd ${WORKDIR}/LightFTP/Source/Release && \
105    cp ${SNPSFUZZER}/tutorials/lightftp/fftp.conf ./ && \
106    cp ${SNPSFUZZER}/tutorials/lightftp/ftpclean.sh ./ && \
107    chmod 777 ftpclean.sh && \
108    cp -r ${SNPSFUZZER}/tutorials/lightftp/certificate /home/
       ubuntu && \
109    mkdir /home/ubuntu/ftpshare
110
111 # Download and compile LightFTP for coverage analysis
112 RUN cd $WORKDIR && \
113    git clone https://github.com/hfiref0x/LightFTP.git LightFTP
       -gcov && \
114    cd LightFTP-gcov && \
115    git checkout 5980ea1 && \
116    patch -p1 < ${WORKDIR}/gcov.patch && \
117    cd Source/Release && \
118    make CFLAGS="-fprofile-arcs -ftest-coverage" CPPFLAGS="-
       fprofile-arcs -ftest-coverage" CXXFLAGS="-fprofile-arcs -
       ftest-coverage" LDFLAGS="-fprofile-arcs -ftest-coverage"
       clean all $MAKE_OPT
119
120 # Set up LightFTP for fuzzing
121 RUN cd ${WORKDIR}/LightFTP-gcov/Source/Release && \
122    cp ${SNPSFUZZER}/tutorials/lightftp/fftp.conf ./ && \
123    cp ${SNPSFUZZER}/tutorials/lightftp/ftpclean.sh ./ && \
124    chmod 777 ftpclean.sh
125
126
127 COPY --chown=ubuntu:ubuntu in-ftp ${WORKDIR}/in-ftp
128 COPY --chown=ubuntu:ubuntu ftp.dict ${WORKDIR}/ftp.dict
129 COPY --chown=ubuntu:ubuntu cov_script.sh ${WORKDIR}/cov_script
130 COPY --chown=ubuntu:ubuntu run.sh ${WORKDIR}/run
131 COPY --chown=ubuntu:ubuntu clean.sh ${WORKDIR}/ftpclean
```

```
132
133
134 # For debugging purposes
135 USER root
136 RUN apt-get -y install ftp
137 USER ubuntu
```

A.2 Modifications ProFuzzBench Scripts

```
1  #!/bin/bash
2
3  FUZZER=$1      #fuzzer name (e.g., aflnet) -- this name must
       match the name of the fuzzer folder inside the Docker
       container
4  OUTDIR=$2      #name of the output folder
5  OPTIONS=$3     #all configured options -- to make it flexible,
       we only fix some options (e.g., -i, -o, -N) in this script
6  TIMEOUT=$4     #time for fuzzing
7  SKIPCOUNT=$5   #used for calculating cov over time. e.g.,
       SKIPCOUNT=5 means we run gcovr after every 5 test cases
8
9  strstr() {
10   [ "${1#*$2*}" = "$1" ] && return 1
11   return 0
12 }
13
14 #Commands for afl-based fuzzers (e.g., aflnet, aflnwe)
15 if [[ $FUZZER = "aflnet" ]] || [[ $FUZZER = "aflnwe" ]] || [[
       $FUZZER = "SNPSFuzzer" ]]; then
16
17   # Run fuzzer-specific commands (if any)
18   if [ -e ${WORKDIR}/run-${FUZZER} ]; then
19     source ${WORKDIR}/run-${FUZZER}
20   fi
21
22   TARGET_DIR=${TARGET_DIR:-"LightFTP"}
23   INPUTS=${INPUTS:-"${WORKDIR}/in-ftp"}
24
25   #Step-1. Do Fuzzing
26   #Move to fuzzing folder
27   cd $WORKDIR/${TARGET_DIR}/Source/Release
28   echo "$WORKDIR/${TARGET_DIR}/Source/Release"
29   if [ $FUZZER == "SNPSFuzzer" ]; then
30     timeout -k 0 --preserve-status $TIMEOUT /home/ubuntu/${
       FUZZER}/afl-fuzz -d -i /home/ubuntu/SNPSFuzzer/tutorials/
       lightftp/in-ftp -x /home/ubuntu/SNPSFuzzer/tutorials/
       lightftp/ftp.dict -o $OUTDIR -N tcp://127.0.0.1/2200
       $OPTIONS -c ${WORKDIR}/ftpclean  ./fftp fftp.conf 2200
31   else
32     timeout -k 0 --preserve-status $TIMEOUT /home/ubuntu/${
       FUZZER}/afl-fuzz -d -i ${INPUTS} -x ${WORKDIR}/ftp.dict -o
       $OUTDIR -N tcp://127.0.0.1/2200 $OPTIONS -c ${WORKDIR}/
       ftpclean ./fftp fftp.conf 2200
```

```
  fi
  STATUS=$?

  #Step-2. Collect code coverage over time
  #Move to gcov folder
  cd $WORKDIR/LightFTP-gcov/Source/Release

  #The last argument passed to cov_script should be 0 if the fuzzer is afl/nwe and it should be 1 if the fuzzer is based on aflnet
  #0: the test case is a concatenated message sequence -- there is no message boundary
  #1: the test case is a structured file keeping several request messages
  if [ $FUZZER = "aflnwe" ]; then
    cov_script ${WORKDIR}/${TARGET_DIR}/Source/Release/${OUTDIR}/ 2200 ${SKIPCOUNT} ${WORKDIR}/${TARGET_DIR}/Source/Release/${OUTDIR}/cov_over_time.csv 0
  else
    cov_script ${WORKDIR}/${TARGET_DIR}/Source/Release/${OUTDIR}/ 2200 ${SKIPCOUNT} ${WORKDIR}/${TARGET_DIR}/Source/Release/${OUTDIR}/cov_over_time.csv 1
  fi

  gcovr -r .. --html --html-details -o index.html
  mkdir ${WORKDIR}/${TARGET_DIR}/Source/Release/${OUTDIR}/cov_html/
  cp *.html ${WORKDIR}/${TARGET_DIR}/Source/Release/${OUTDIR}/cov_html/

  #Step-3. Save the result to the ${WORKDIR} folder
  #Tar all results to a file
  cd ${WORKDIR}/${TARGET_DIR}/Source/Release
  tar -zcvf ${WORKDIR}/${OUTDIR}.tar.gz ${OUTDIR}

  exit $STATUS
fi
```

A.3 Grammar model BooFuzz

```
from boofuzz import *

session = Session(sleep_time=2, target=Target(
    connection=SocketConnection("127.0.0.1", 2200, proto='tcp')
    ))

user = Request("user", children=(
    String("key", "USER"),
    Delim("space", " "),
    String("val", "USER fouad"),
    Static("end", "\r\n"),
))

```

```
passw = Request("pass", children=(
    String("key", "PASS"),
    Delim("space", " "),
    String("val", "PASS fouad"),
    Static("end", "\r\n"),
))

stor = Request("stor", children=(
    String("key", "STOR"),
    Delim("space", " "),
    String("val", "AAAA"),
    Static("end", "\r\n"),
))

retr = Request("retr", children=(
    String("key", "RETR"),
    Delim("space", " "),
    String("val", "AAAA"),
    Static("end", "\r\n"),
))

session.connect(user)
session.connect(user, passw)
#session.connect(passw, stor)
#session.connect(passw, retr)

session.fuzz()
```