YRKESHÖGSKOLAN NOVIA

Evaluation of Fuzzing as a Test Method for an Embedded System

Thomas Wilson

Bachelor's thesis

Electrical and Automation Engineering

Vaasa 2018

BACHELOR'S THESIS

Author:	Thomas Wilson
Degree Programme:	Electrical and automation engineering, Vaasa
Specialization:	Automation
Supervisor:	Roger Mäntylä

Title: Evaluation of Fuzzing as a Test Method for an Embedded System

Date April 19, 2018 Number of pages 43 Appendix 1

Abstract

When creating software, the stability, quality, and security of the software is an important part of the development. To ensure good quality, some form of software testing must be used.

This thesis evaluates the use of fuzzing as a software testing method. The basic idea of fuzzing is to, in a semi-random way, manipulate input data to a program in an attempt to make the program crash or misbehave, and thereby find faults in the software. The theory of fuzzing and different types of fuzzing will be described. The focus is set on fuzzing to improve the quality of the software, rather than the security aspect.

Methods used to evaluate fuzzing as a test method include; finding what kind of fuzzing tools that are available today, and implementation of one fuzzing method. For the implementation, the fuzzing tool American Fuzzy Lop (AFL) was chosen to do tests on an embedded systems application.

The results are that fuzzing can be used as a test method for an embedded system. But the benefits of fuzzing can vary, depending on the system tested. In the tests that were made only low hanging fruit, shallow faults, could be found.

Language: English Key words: fuzzing, American fuzzy lop, software testing

EXAMENSARBETE

Författare:	Thomas Wilson
Utbildning och ort:	El- och automationsteknik, Vasa
Inriktningsalternativ:	Automation
Handledare:	Roger Mäntylä

Titel: Evaluering av fuzzing som en testmetod för ett inbyggt system

Datum 25.4.2018 Sidantal 43 Bilaga 1

Abstrakt

Kvalité, stabilitet och säkerhet är viktiga faktorer som måste beaktas när en mjukvara utvecklas. För att säkerställa att alla faktorer uppfylls, så måste mjukvaran genomgå någon form av testning.

I denna avhandling undersöktes mjukvarutestning med metoden fuzzing. Grundidén med fuzzing är att, på ett delvis slumpartat sätt, manipulera data som skickas till ett program för att se hur programmet reagerar. Om programmet då kraschar, eller på annat sätt utförs felaktigt, så har ett problem hittats i mjukvaran. I avhandlingen beskrevs teorin bakom fuzzing, såväl som olika typer av fuzzing. Fokus låg på fuzzing i syfte att öka kvalité och stabilitet, och mindre på säkerhetsfaktorn.

Metoder som använts för att evaluera fuzzing som testmetod inkluderar; undersökning av vilka fuzzing verktyg som finns tillgängliga idag, och implementering av en fuzzing-metod. Till implementeringen valdes fuzzing verktyget American Fuzzy Lop (AFL), för att utföra tester på en applikation för ett inbyggt system.

Resultatet var att fuzzing är en metod som kan användas för ett inbyggt system, men beroende på systemet så kan fördelarna med fuzzing variera. I de utförda testerna så kunde endaste ytliga problem påträffas.

Språk: engelska Nyckelord: fuzzing, American fuzzy lop, mjukvarutestning

Contents

1 Introduction

It is nearly impossible to write a big complex program without making some small mistakes, introducing bugs and possible vulnerabilities into the code. Finding bugs, causes of program crashes, or vulnerabilities is a big challenge. There are many different approaches to tackle these, and for a good result, more than one approach must be used. A relatively unknown method to approach these problems is called fuzzing. Fuzzing is quite a different approach compared to the commonly used methods. Fuzzing is about testing the limits of inputs made to the system under test (referred to as SUT), by automatically feeding the program with semi-random or mutated data.

This thesis is going to explore what fuzzing is, how it can be used, what software is available and evaluate how fuzzing could be applied to embedded systems at Wärtsilä.

1.1 Wärtsilä

This thesis is written for Wärtsilä Finland, department of Engine Performance and Control. Wärtsilä is a Finnish company, established in 1834. Today Wärtsilä is a technology company delivering the world's most efficient engine, complete systems for ships, power plants, and maintenance for all their systems. One in three ships around the world run with Wärtsilä technology. In 2016, Wärtsilä employed 18,000 people with operations in more than 200 locations in 70 countries. Wärtsilä is listed on Nasdaq Helsinki and in 2016 had net sales of 4.8 billion euro. Wärtsilä is divided into three main divisions, Marine Solutions (35%), Energy Solutions (20%), and Services (46%), (% of net sales by area – 2016). [1]

1.2 Background

An essential part of software development is to test and verify that the written software works as expected, even under unusual conditions. Testing involves finding bugs, unexpected behaviors, and vulnerabilities. For this, there needs to be more thorough testing than only testing that the functionalities meet the specifications.

There is no single answer to what the best method for discovering bugs and vulnerabilities in software is. There are a bunch of different approaches to software testing, all with their individual strengths; there is no one method that for sure will find all the bugs and vulnerabilities in a given software. Different approaches will uncover different types of problems. To get the best possible coverage, a mix of different methods are necessary [2].

1.3 Purpose and delimitations

The purpose of this thesis is to evaluate the use of fuzzing as a test method on an embedded system to increase the software quality.

The scope will be delimited to fuzzing of an embedded system with a focus on the programming languages c and c++, compiled by gcc and g++. A few fuzzers will be chosen and evaluated at a high level, one of them is then used for implementation testing. There will be no extensive testing or comparison of performance between fuzzers.

1.4 Method

Different fuzzing methods will be described and evaluated for use on an embedded system. Google will be used to find available fuzzing tools, fuzzers. A few useful fuzzers will be considered, and one will be chosen for further testing. Tests will be made on a small program to get an idea of how fuzzing can be implemented, and find out which type of problems that may occur.

2 Theory

This chapter explains what software testing is, before going deeper into fuzzing in the next section. Discussed here are; bugs and vulnerabilities, the concept of code coverage, and the differences between black-, white-, and grey-box software testing.

2.1 Bugs and vulnerabilities

A bug is when a system is not behaving the way it is intended to. Can also be recognized as a fault or defect. There are many ways that a bug can appear in a program, like faulty logic, access errors, or executing code with undefined behavior such as a buffer overflow. Buffer overflows can occur when an array is miscalculated somehow, this type of bug should be found by fuzzing. A vulnerability is a bug that in some way can introduce an opportunity for malicious use of the system. Bugs need to be examined to determine if they also can be a vulnerability, open to intruders.

2.2 Code coverage

In software testing, code coverage is often mentioned. Code coverage is the percentage of the total amount of code in a software, that is being tested by some testing method. Although 100% code coverage should not be strived for [3], it can be used as a measure for how well tested the program is. High code coverage can be used as proof that the software is working as intended. Fuzzing can be used in many ways to gain a good code coverage or increases the code coverage of existing tests.

2.3 Software testing

The main categories in software testing are black, grey, and white box testing. The difference among these categories is determined by what level of information that is available to the tester. At one end is white-box which requires complete access to all the resources, source code, and specifications. At the other end is black-box, which requires no knowledge of the internal specifications and therefore is more of a blind test. In the middle is grey box testing. Grey-box can have many definitions depending on the situation, but it involves the tester having access to some additional information of the SUT, like compiled binaries. Fuzzing falls mostly into the grey box area because there are so many different approaches to fuzzing that it can cover the whole spectrum from white to black box. [2] This thesis involves testing of both white- and black-box fuzzing of software.

2.3.1 White-box testing

An example of a pure white box testing is a source code review. A source code review can be done manually, line by line, but that is not practical to do on more extensive programs. The assist of automated tools is needed to go through all the code to find suspicious code, potentially bugs. Automated source code analyzers can detect a lot of potential risks that all need to be analyzed to decide if they are a real problem or not. [2] Fuzzing can be used as a white-box test to automatically test code to find real problems. That is when fuzzing is used on a system while using the source code to provide the fuzzer with internal information about the progress of the fuzzing.

2.3.2 Black-box testing

Black-box testing is done when the tester has a program, ready to use, where inputs can be made to the so-called black-box, and the output can be observed. In other words, black-box is how all conventional software is used by the end user. There is no knowledge of the inner workings of the black box. [2]

Fuzzing can always be used as a black-box test if there is a way for the fuzzer to provide inputs to the program. Black box fuzzing is considered blind fuzzing; the fuzzer does not have any information about what is happening inside the application.

2.3.3 Grey-box testing

Grey box testing is right in between white and black box. Grey box testing is performed when you have some more in-depth knowledge of the system than the plain black box, but don’t have access to the source code or no purpose to do a white box test. Grey box testing may involve a black box with the additional reverse engineering of the binary. [2]

Grey box fuzzing can be performed if some manipulation is done to the black-box binary. Some fuzzing tools can insert instrumentations into a pre-compiled binary, providing information to the fuzzer.

3 Fuzzing

Fuzzing is an automated method for inserting unexpected, mutated, inputs to a system to test how well the system handles unexpected inputs. Fuzzing is about finding inputs that results in faults or undefined behaviors in the system under test (SUT). [2] By inserting unexpected inputs, new unforeseen or rare code paths can get triggered, possibly finding unknown bugs, so-called zero-days. To find new code paths and bugs, some kind of random generator generates or mutates inputs to a system. A massive number of inputs is usually required to get any results; therefore automation must be used to feed new data as fast as possible. [4] If any faults are found during the fuzzing, the input causing the failure should be automatically saved. The faulty input can then be tested and analyzed to see precisely why that input causes problems.

Fuzzing complements other testing methods, increasing the code coverage. Even when a system has been tested with good test cases, fuzzing can trigger new, unexpected execution flows. Fuzzing often finds bugs in corner cases, right at the edge between allowed and unallowed inputs, that otherwise may be hard to detect. Fuzzing should be used to test every input of the system, especially those accepting inputs from the internet or from files. [4]

3.1 History of fuzzing

Barton Miller is seen as the founder of the word fuzz testing and fuzzing, as Miller one night in 1988 realized how fragile programs could be:

"Sitting in my apartment in Madison in the Fall of 1988, there was a wild Midwest thunderstorm pouring rain and lighting up the late night sky. That night, I was logged on to the Unix systems in my office via a dial-up phone line over a 1200 baud modem. With the heavy rain, there was noise on the line and that noise was interfering with my ability to type sensible commands to the shell and programs that I was running. It was a race to type an input line before the noise overwhelmed the command.

This fighting with the noisy phone line was not surprising. What did surprise me was the fact that the noise seemed to be causing programs to crash. And more surprisingly to me was the programs that were crashing – common Unix utilities that we all use everyday." [5] [6]

After that night Barton Miller added a project to his class at the University of Wisconsin. The project was to make an automated script feeding random data to common Unix programs and make them crash. He settled on the term "fuzz" to describe the randomness of the testing method. [5]

Since then fuzzing has grown into a technique used by both hackers, cybersecurity, and software developers. The idea of fuzzing is to find bugs, by giving unexpected inputs to a program. A software developer might want to fix the bugs, to make a more robust software less likely to crash. Cybersecurity might want to know if the bugs are a potential security concern that needs to be fixed. A hacker might try to exploit a bug before it is fixed. It is a matter of who finds the bug first. Fuzzing has become a popular alternative since it can be applied to virtually any layer of the software. Many good open source fuzzing programs are available, and a few commercial ones.

3.2 Implementing fuzzing

Like other testing methods for software projects, fuzzing requires planning and maintenance to keep the fuzzing compatible with the system. Before beginning fuzzing on a project, several questions need to be evaluated and answered to make sure that the fuzzing will be successful.

What kind of system is the target for the fuzzing? Fuzzing can be applied to any type of input that the SUT consumes. Inputs could be file input, network input, command line input, etc. What kind of problems are expected to be found? Fuzzing can be done for both quality and stability, and for cybersecurity. How big is the fuzzing project going to be? Smaller projects can be fuzzed on a local computer, while bigger projects would need a dedicated machine.

3.2.1 Choose a target

Before beginning to fuzz, different targets should be considered. Interesting targets for a fuzzer are programs, parsers, or libraries accepting external inputs. Inputs can be through external interfaces, like files or network connection, or internal functions in a library. [4] The availability of source code, or compiled binaries, should be considered, to know if a black box or white/grey box method can be used. The target should be the newest version of the software, that is where bugs most likely are found. [7]

Fuzzing of the target can begin faster if there are ready-made basic code examples for the target. If there are existing test cases for the target, like a unit test, these can be a help in getting started on fuzzing. [7]

An interface can contain many layers, often parsing and consistency checks are done before the actual data is handled. Therefore, when fuzzing, the part to be fuzzed needs to be defined.

Otherwise, the fuzzer might get stuck fuzzing one part, never reaching the interesting parts. [4]

When fuzzing, speed is essential to get results in a decent time. A modern fuzzer can often do several thousands of test cycles per second. With access to the source code, there are several ways to speed up the fuzzing process. A program can be modified to bypass overhead, and pass fuzzed data directly to the target functionality. CRC checks, unnecessary logging, file input/output, or other time-consuming tasks can be disabled. This can be defined as a fuzz-friendly build of the SUT. But when altering the SUT, caution must be taken not to hide or create any bugs. Optimization should not be a priority. When a fuzzing system is up and running, ways to make optimizations can be considered. [4] Any bugs found should later be verified on an unaltered system.

3.2.2 Scalability

There are different ways to implement fuzzing to a system. The most straightforward way is a person running a fuzzing instance on his own machine, fuzzing a target program and manually checking if bugs have been found. In large-scale fuzzing, automation becomes essential. Fuzzing can be a part of a CI (Continuous Integration) system, automatically fuzzing different targets of the system, changing target a few times per day. When going large scale, the fuzzing can be executed by a cloud using many computers in parallel to automatically perform fuzzing. With many fuzzers running in parallel things like automatic reporting of findings and automatic minimization of test cases become essential. Before beginning large-scale fuzzing, planning must be done to ensure the usability of the fuzzer with different kinds of targets. [4]

3.2.3 Starting

To start fuzzing, the first thing to do is to define what kind of system that is going to be the target. By looking at the system, a rough list of requirements can be made for the fuzzer. The primary requirements are, what type of input does the system take, how is the input delivered to the system and if or how the system can be instrumented for instrumentation guided fuzzing. When the requirements are ready, it is time to search the internet for available fuzzers. There are a lot of open source fuzzers, freely available, and a few commercial ones. Open source fuzzers can work well in many situations and are readily accessible for anyone to try fuzzing. For an organization wanting to start fuzzing many products fast, a commercial fuzzer can be a good option. [4]

Commercial fuzzers usually deliver the whole package, test cases generation, injection, instrumentation, automation, and a user-friendly interface. They also come with readymade fuzzing solution for common protocols and file types.

Open source fuzzers come in all different combinations of use cases. Some are made for a specific type of target; some are general purpose ready to go for common targets, some are framework-types used for the building of custom frameworks for the target. Some only deliver the test case generation, some also the injection of test cases, and some the full package with instrumentation and automation.

3.2.4 Monitoring

To know how the fuzzing is proceeding, the SUT must be monitored. The monitor provides information both to the fuzzer to improve the testing, and to the user to show statistics of how the fuzzer is doing. That leads to the big question of when to stop fuzzing a target. If the fuzzer has not found any new bugs for a relatively long time, it might be a good time to stop. The fuzzer could be stuck, not getting further, but at the same time, a new bug may be found if letting the fuzzer go for a while longer. With a coverage-guided fuzzer, it is a bit easier. As long as new code paths are being found, the fuzzer is doing good. If no new code paths have been found for a while, it is time to consider if the fuzzer is done. Either there are no more paths to be found in the target, with the current setup and environment, or the fuzzer is stuck at some hard to reach code path. If the SUT is built from source code, then fake bugs can be added to aid in tracking of the fuzzer. To verify that the fuzzer is traversing the code as expected, a print and abort function can be added at interesting points in the code. Make sure that the fake bugs do not block the fuzzer in any way. [4] When crashes have been found they can manually be further analyzed, with a software debugging tool, to trace precisely what is causing the crash.

3.2.5 Maintenance

For the fuzzing to be effective in the long run, maintenance is required. [4] When the fuzzer is stopped, it does not mean it is done fuzzing. There could still be bugs to discover. When fuzzing a new target, finding many new test cases, it is a good idea to stop the fuzzer at some point for maintenance. Depending on the fuzzer, a lot of good test cases could be saved by the fuzzer that can be analyzed and minimized to remove redundant test cases. These can then be run back to the fuzzer to make the fuzzing more efficient. [7]

3.2.6 Flexibility

When choosing a fuzzer, the flexibility of it should be considered so that it can easily be adjusted for changes of the SUT. A flexible fuzzer is a fuzzer that requires no or little, configuration changes to target different types of systems. A lot of time is wasted if the fuzzing system must be remade when the SUT changes. A flexible fuzzer is also useful in the remark that different fuzzing approaches can find different issues. A flexible fuzzer can also support switching of components, such as instrumentation options. [4]

3.3 Fuzzing methods

There are some basic fuzzing methods, used in different variations by fuzzers. Fuzzers implements one or more methods to achieve the best results. The methods can be purely random, mutational, generational, or evolutionary which is based on both mutation and generation.

3.3.1 Pure random

An example of the most basic fuzzing method, yet sometimes effective:

```
while[1]; do cat /dev/urandom | nc -vv target port; done
```
[2]

This command consists of an infinite loop streaming pseudo-random data to a target address and port. This simple random method has found faults in real software. The problem is to find which random data caused the fault, and the code coverage is shallow. Still, this is an easy way to test the basic robustness of a program, and demonstrates what fuzzing is. [2]

3.3.2 Mutation

A mutation based fuzzer is a brute force fuzzer, often called a dumb fuzzer. Little to no knowledge is needed about the SUT, file format or protocol. Brute force refers to the fuzzer taking in a valid sample of data, sequentially modifying every byte, word, string, etc., and sending that data to the target. [2]

The fuzzer is improved by implementing some intelligence, like allowing parsing of the samples before mutation, to make sure that only specific parts of the data are mutated, not breaking the overall structure. [8]

It is a very straightforward method, although inefficient as is. By somewhat randomly modifying parts of the data it will take many iterations to get any results. Code coverage is dependent on having many good data samples, exercising different parts of the code because the mutations alone is unlikely to stumble into the deeper, more complex code paths. [2] Mutation is the core fuzzing algorithm behind many open source fuzzers, using smart algorithms to reach more complex code paths than mutation alone would. A code example of mutations can be found in the appendix.

3.3.3 Generation

Often called a smart fuzzer since generation requires knowledge of the data structure [8]. A generation based fuzzer generates inputs, test cases, based on a given specification of a file format or protocol. The goal of the generation is to generate test cases, containing some invalid data causing problems in the SUT, but still valid enough to not get instantly rejected. [9] Since generational fuzzers need specifications of the file format or protocol to be fuzzed, a framework of the specifications needs to be made. Creating a useful framework takes time, but can result in very efficient fuzzing. Commercial fuzzers are mostly based on generation, delivering readymade frameworks for all the common formats and protocol; offering to create custom frameworks for specific targets.

Generation fuzzers often take a file format, or protocol and split it up into chunks. This allows single chunks to be fuzzed independently while keeping the overall structure of the data format. This makes the fuzzer able to get deeper into the SUT as it can generate sequences of valid inputs to specific parts of the communication. By knowing the specifications of the protocol used, the fuzzer is also able to give dynamic responses to communications when acting as a client/server. [8]

3.3.4 Evolutionary

Evolutionary fuzzers are the latest breed of fuzzers; they began to appear around the year 2007 [10]. An evolutionary fuzzer uses advanced algorithms to get better code coverage, than plain mutation or generation. The evolutionary fuzzer still relies on either mutation or generation as the core functions for fuzzing the test cases. It uses some form of instrumentation of the binary to get information and guidance, to make smarter mutations to reach deeper into the SUT. By instrumenting the binary, the evolutionary fuzzer can track the code coverage of different test cases. [8] A white box fuzzer can insert instrumentations during the compilation of the SUT. There is also evolutionary fuzzers not dependent on the source code. Instead, a grey-box approach can be used, debugging the target binary for code coverage and data. [10]

By looking at the code coverage, the fuzzer can find out which part of the test case leads to which part of the code. The fuzzer gradually evolves, creating new test cases, covering increasingly more of the program code, with the goal of trying to find test cases with the best possible code coverage. [8] The test cases discovered by the evolutionary fuzzer is also a good source for test cases to be used in other situations.

Evolutionary fuzzers often relies on genetic algorithms. [8] There are two types of algorithms used in evolutionary fuzzers, evolution strategies, and genetic algorithms. A combination of these can be used. Both try to generate the optimal test cases based on the previous generation of test cases. Evolution strategies use mutation, modifying parts of a good test case to create new ones. Genetic algorithms use two or more good test cases, cut them apart, combining them into new test cases. At that stage, a bit of mutation can be added to get more diversity in the test cases. The evolutionary fuzzer can then give different test cases a type of fitness score depending on metrics like code coverage and code depth. This score is used to determine if a test case should continue to the next stage of the evolution. [11]

The author of AFL, Michal Zalewski, made an interesting demonstration of the concept of evolutionary fuzzing by “pulling JPEGs out of thin air”, using AFL. Zalewski made AFL feed a plain text file with the word “hello” to a simple JPEG image viewer. This is not how it is meant to be used since the text file was immediately rejected by the image parser. By observing binary instrumentations, the AFL fuzzer found the correct bytes for the header of a valid JPEG. Still not valid file, but the instrumentations showed that new code paths were triggered, indicating the fuzzer that is was on the right track. Using the header as seed, more parts of the JPEG structure was found. About six hours later the first entirely valid, grey-scale, image was created. That was done on an 8-core system. But JPEG is still a simple file type, anything more complex would require a better starting test case to get any results in a decent time. [12] This is an example of how evolutionary fuzzers can be used without much, or any at all, knowledge of the file structure being fuzzed.

3.4 Types of fuzzing

Since fuzzing can be done on many kinds of targets and layers, different fuzzing types can be defined to separate one fuzzing from another. On a high level, fuzzing can be divided into local fuzzing or remote fuzzing.

3.4.1 Local

Local fuzzing is performed on a local system that parses user inputs either directly or through a file. Local fuzzing can then be split into file format fuzzing or environment variable and command line fuzzing. File format fuzzing aims at a specific target's ability to parses information from a file, trying to find flaws in the parsing. Similarly, a target can be fuzzed through command line arguments or environment variables [2]. This type of fuzzing is the most relevant type for an embedded system. Embedded systems can parse files or commands for, I.e., parameter or configuration inputs.

3.4.2 Remote

Remote fuzzing can either be seen on a general level, as network fuzzing, or a more specific level as a web application or server fuzzing. Network fuzzing aims at fuzzing a target over a network protocol. The target is fuzzed by mutating or generating fuzzed values, that are communicated over to the target through a network protocol while monitoring the behavior and response of the target. Network fuzzing tends to either target a specific protocol or be a generic framework type where the user needs to build a framework for the specified protocol. Due to the network protocol creating overhead, network fuzzing is considerably slower than local fuzzing. [2] On an embedded system, network fuzzing could be used if a network protocol is used by the system for any external or internal communication.

Web application and server fuzzing. Web application fuzzing is a specific type of network protocol fuzzing, specialized on fuzzing HTTP packets for web applications. [2] Web applications is a common type of network fuzzing, and since can have its own category. On an embedded system, this type of fuzzing applies if the SUT hosts a web server.

3.5 Limitations

Fuzzing has a limited ability to find certain types of faults. Fuzzing is generally good at finding low hanging fruit, easy bugs, that make the program crash or hang. More complex bugs can be hard to detect with fuzzing. If the SUT contains checksums or encryption, a brute-force fuzzer may get problems. A way to work around that is to disable those parts of the SUT [13]. A fuzzer cannot know if it somehow gained access to a backdoor allowing it to use administrator only code parts. If there is poor logic in the system, without resulting in any execution fault, then the fuzzer cannot detect that. If there is memory corruption but the system is made to handle them in such a way that the program exits without faults, then the fuzzer usually can’t detect that. [2] Fuzzing may also trigger other silent bugs such as, memory leaks, excessive CPU load, memory corruption, command injections, etc. [4] There are some tools available that can help the fuzzer in finding these kinds of problems, but they may be hard to apply, and slowing down the fuzzing process.

4 Fuzzers

A fuzzer is the program that is performing the fuzzing, or fuzz testing. A typical fuzzing approach is described by the six steps listed below. A fully featured fuzzer should be able to do step 3, 4, 5, and aid in step 6. The rest is up to the user.

1. Choose a target

The target for the fuzzer can be a program, or any part of a system that is parsing inputs.

2. Identify inputs

Possible inputs to the SUT must be identified. Any inputs to the target should be considered, files, filenames, headers, variables, etc. Priority should be on user inputs.

3. Generate data

The fuzzing tool automatically either generate fuzzed data or mutate existing data as inputs to the target.

4. Execute

The fuzzing tool automatically sends the fuzzed data to the target and/or executes the target application with the fuzzed data.

5. Monitor

While fuzzing, the target must be monitored for any exceptions. If the target crashes, the fuzzer must know exactly which input data caused the crash.

6. Analyzing the results

Depending on the goal of the fuzzing, the results, crashes, can be used in different ways. The data causing the crash should be tested and confirmed, to determine if the faults can be reproduced in a real situation, and needs to be fixed. If security is concerned, a security expert should decide if the fault could be exploitable. [2]

4.1 Comparison factors

To evaluate fuzzing, different fuzzing software available need to be examined. To make a comparison of the fuzzers, a small set of essential factors are highlighted for each fuzzer. Below is a model-list of the comparison factors used.

Availability	Open source / free / commercial
Support	Is the fuzzer still alive and in active development? Does the fuzzer have an active community of users? Forums?
Operating system	Linux / Windows
System requirements for feedback-driven (white-/grey-box) fuzzing mode	Software compiler supported, or hardware required
Type	File / variable / protocol / network / web Mutation / generation

To find available fuzzers google was used to search for keywords. Open source fuzzers are the most common, but many projects have not been updated in several years, and thus focus is on finding up to date and active fuzzers. Here is a list of the most promising fuzzers that were found, that are up to date and in active development.

4.2 American Fuzzy Lop (AFL)

Availability	Open source
Support	Active development, and a mailing list (forum) with a few user posts per week.
Operating system	Linux
System requirements for feedback-driven (white-/grey-box) fuzzing mode	Source code compiled with GCC or Clang.

Type	Evolutionary file fuzzer, based on coverage-guided mutations.

AFL is an open source evolutionary file-fuzzer using genetic algorithms. To use AFL's full potential, the source code for the SUT needs to be available, although it can be used without source code as a black-box test. AFL has its own compilers, that is used to compile the program while inserting small pieces of code into the program allowing AFL to track all the code paths in the program. Although AFL tracks code coverage, it is not shown in a human-readable way. To monitor actual code coverage an external tool, afl-cov, can be used. [13]

AFL is an instrumentation-guided genetic fuzzer, capable of synthesizing a wide range of file inputs for targets. It is built around mutation of test cases that have been found to generate the most efficient results. AFL has a user-friendly interface and requires almost no adjustments by the user [14] AFL works best with programs written in C or C++ [7].

4.3 Honggfuzz

Availability	Open source
Support	Active development, inactive mailing list.
Operating system	Linux / Windows / OS X / Android
System requirements for feedback-driven (white-/grey-box) fuzzing mode	Software compiled with Clang or GCC. Or an Intel CPU with support for BTS or PT process tracing.
Type	Evolutionary, coverage-guided, mutational, in-process fuzzer.

Honggfuzz is an easy to use, evolutionary, coverage-guided, fuzzer. It is multi-threaded, to automatically utilize available processing power. Honggfuzz can run white-, black-, or grey-box fuzzing. For process tracing, either software or hardware can be used. Software tracing is done by instrumenting the binary during compilation, like AFL, while hardware tracing is supported on a relatively new Intel CPU with support for the process tracing techniques BTS, or PT. [15]

Uses persistent in-process fuzzing, repeatedly calling an input function with fuzzed data, without restarting the SUT. If in-process fuzzing cannot be used, the more standard fuzzing mode can be used where the process is restarted after every input, but this is slower than the in-process mode. [15]

4.4 LibFuzzer

Availability	Open source, part of the LLVM suite
Support	Active development, and a mailing list (forum) with a few user posts per week.
Operating system	Linux / Windows
System requirements for feedback-driven (white-/grey-box) fuzzing mode	Source code compiled with clang
Type	Evolutionary, in-process, coverage-guided
Other	Originally inspired by AFL, therefore containing many similarities

LibFuzzer is an in-process library fuzzer, closely inspired by AFL. [14] It is an evolutionary coverage-guided fuzzer for fuzzing libraries and functions. LibFuzzer is provided as part of the LLVM suite. Clang is used for compilation as binary coverage information is provided by LLVM’s SanitizerCoverage binary instrumentation. [16]

In-process fuzzing relies on executing the target function inside a loop. This is faster than running the whole program for every new input or using fork servers (as standard AFL does). When using in-process fuzzing, the program could crash because of memory corruption from previous inputs. Address sanitizer is used to detect and handle memory corruptions. [16]

4.5 Peach fuzzer

Availability	A commercial edition, and a free open source edition
Support	Commercial edition comes with dedicated support. The open source edition has e support forum, which is very inactive.
Operating system	Linux / Windows / OS X
System requirements for feedback-driven (white-/grey-box) fuzzing mode	-
Type	Generational framework fuzzer

Peach fuzzer is a commercial framework-type, generational, fuzzer. They also deliver Peach Community which is a free to use, open source, version. The open source edition has fewer features and not much development. The commercial edition is based on the original open source edition, with many improvements. The generational framework type fuzzer requires the building of a custom framework for the SUT. The framework explains the structure of the data that is being fuzzed so that the fuzzer know which parts should be mutated and which should not. Building a framework can be effective for fuzzing a specific target. The commercial version comes loaded with pre-made frameworks for all kinds of standard protocols and data structures. [17]

4.6 Defensics Fuzz Testing

Availability	Commercial
Support	Official support and an active support forum
Operating system	Linux / Windows / OS X
System requirements for feedback-driven (white-/grey-box) fuzzing mode	-

Type	Generational framework fuzzer

Synopsys Fuzz Testing, Defensics, also known as Codenomicon Defensics is a commercial generation based fuzzer. It delivers many pre-made test suites for protocols and file formats. It also provides a fuzzing framework for testing of custom file formats or protocols. [18]

4.7 Burp Suite

Availability	Fully featured commercial edition, 329€ / year / user. Free edition available, limited in speed and features. A free 30-day trial for business users.
Support	Support center available on the software's homepage, with lots of information and tutorials. Active community forum.
Operating system	Windows / Linux
System requirements for feedback-driven (white-/grey-box) fuzzing mode	-
Type	Web application scanner with web fuzzing capabilities

Portswigger's burp suite is a platform for fuzzing and security testing of web applications. The suit includes many different tools for security testing. There are two versions available of Burp Suite, Professional, and community. Professional consists of the full platform, for a yearly price per user. A free trial period of 30 days is available for business users. The free community edition contains most of the tools, but with very limited speed. [19]

5 Implementation

Fuzzing can be implemented in a wide range of situations. Black-, grey-, and white box. When fuzzing is implemented as a black box test, it is considered blind with no details of the internal state of the target [2]. Grey and white box fuzzers can get additional information by monitoring the internal state of the target, such as code coverage, to increase the efficiency of the fuzzing.

Implementation tests will be made with a file fuzzer, on programs reading inputs from files. Both black- and white-box approaches will be tested, to verify what the differences are. All testing will be executed inside a virtual Linux machine.

5.1 Choosing test method

To evaluate fuzzing as a test method, American Fuzzy Lop (AFL) was chosen for implementation testing. AFL was selected since it is a well-suited fuzzer for the type of system that is being tested; a local system reading inputs from a file, with source code available to build from. It is an easy to use, well tested and documented fuzzer, allowing for more time spent on testing, and less on setting up the fuzzer. To give a clearer understanding of how a fuzzer works, a small test program was made for testing. The test program was tailored for use with a file-fuzzer, like AFL, to make initial tests before running AFL on the real system.

5.2 Using AFL

AFL has a few modes and options that can be used, but in these tests, the most standard setups will be used. Per the AFL manual [13] the following steps are taken to start a basic fuzzing session with AFL;

1. To use AFL for fuzzing an application the target needs to receive data from standard input or a file; or modified to do so. For AFL to detect faults, the target must crash properly when encountering a fault.

2. Compile the target using afl-gcc, or afl-g++.

3. Collect one or more, as small as possible, valid input files for the target. These files are the starting point for the fuzzer to mutate from.

4. Start the fuzzing. The basic way of fuzzing a program reading from standard input would be: “./afl-fuzz -i testcase_dir -o findings_dir -- \ /path/to/tested/program [...program's command line...]”

5. Wait for the results.

So, before starting AFL, an input and an output folder should be created. The input folder should be filled with valid test files for the SUT. The output folder is where AFL will store all of its data and findings. When AFL starts, it will read and test all the files in the input folder, to validate them, and then copy the test files to a folder named queue inside the output folder. The files in the queue folder are what AFL is going to use as test cases to mutate during the fuzzing. If during testing, a mutated test case is found to trigger a new path, then that test case is going to be added to the queue folder. Inside the output directory, are also two folders named crash, and hangs. If a test case is found to trigger a crash, or a hang, of the SUT, then that test case is going to be added to the respective folder. A test that triggers a crash, or hangs, is not going to be inserted to the queue.

Below is a basic example of the command line that is going to be used for most of the coming tests.

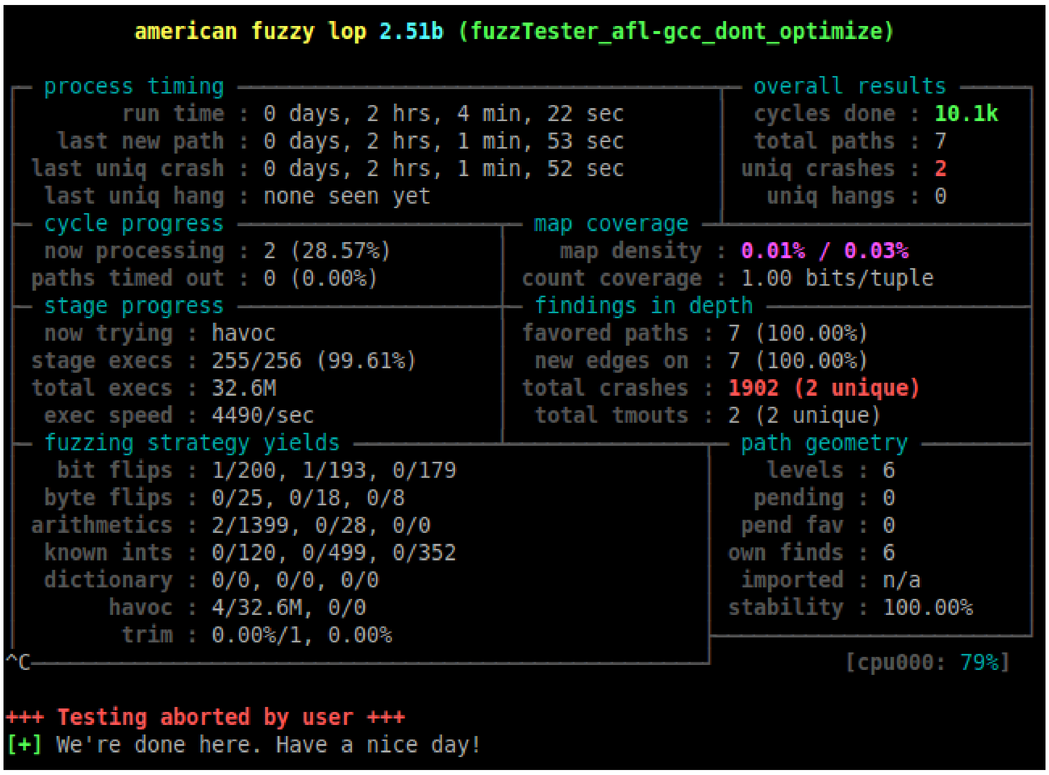
```
afl-fuzz -i ./i/ -o ./o/ ./path/to/SUT [SUT options] @@
```

“-i ./i/” means the input folder is named ‘i’, while “-o ./o/” means the output folder is named ‘o’. Next comes the path to the SUT, possible options (arguments) for the SUT, and finally @@ is the place where the SUT normally would expect a file to be inserted. That is where the fuzzer automatically will insert mutated files.

When AFL is run for the first time, the fuzzer will probably stop with an error message about the handling of core dumps. The file core_pattern, located at /proc/sys/kernel/core_pattern needs to contain the word “core”, for AFL to gain direct access to information about a program crashing. An easy way to edit the file is by using the following command: `sudo sysctl -w kernel.core_pattern=core`

5.2.1 Understanding AFL’s status screen

This section explains the important details of AFL’s status screen, and how to interpret the data.



```
                american fuzzy lop 2.51b (fuzzTester_afl-gcc_dont_optimize)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 2 hrs, 4 min, 22 sec       │  cycles done : 10.1k  │
│   last new path : 0 days, 2 hrs, 1 min, 53 sec       │  total paths : 7      │
│ last uniq crash : 0 days, 2 hrs, 1 min, 52 sec       │ uniq crashes : 2      │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 2 (28.57%)        │    map density : 0.01% / 0.03%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.00 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 7 (100.00%)            │
│ stage execs : 255/256 (99.61%)      │  new edges on : 7 (100.00%)            │
│ total execs : 32.6M                 │ total crashes : 1902 (2 unique)        │
│  exec speed : 4490/sec              │  total tmouts : 2 (2 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 1/200, 1/193, 0/179                   │    levels : 6          │
│  byte flips : 0/25, 0/18, 0/8                       │   pending : 0          │
│ arithmetics : 2/1399, 0/28, 0/0                     │  pend fav : 0          │
│  known ints : 0/120, 0/499, 0/352                   │ own finds : 6          │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 4/32.6M, 0/0                          │ stability : 100.00%    │
│        trim : 0.00%/1, 0.00%                        ├────────────────────────┘
^C────────────────────────────────────────────────────┘          [cpu000: 79%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
```

Figure 1. Example of AFL status screen

The status screen provides lots of information in a compact layout; the essential parts will be explained here. Detailed information about the status screen can be found in the AFL documentation, provided with the source code. At the top of the screen, the version of AFL is shown together with the name of the program that is being fuzzed.

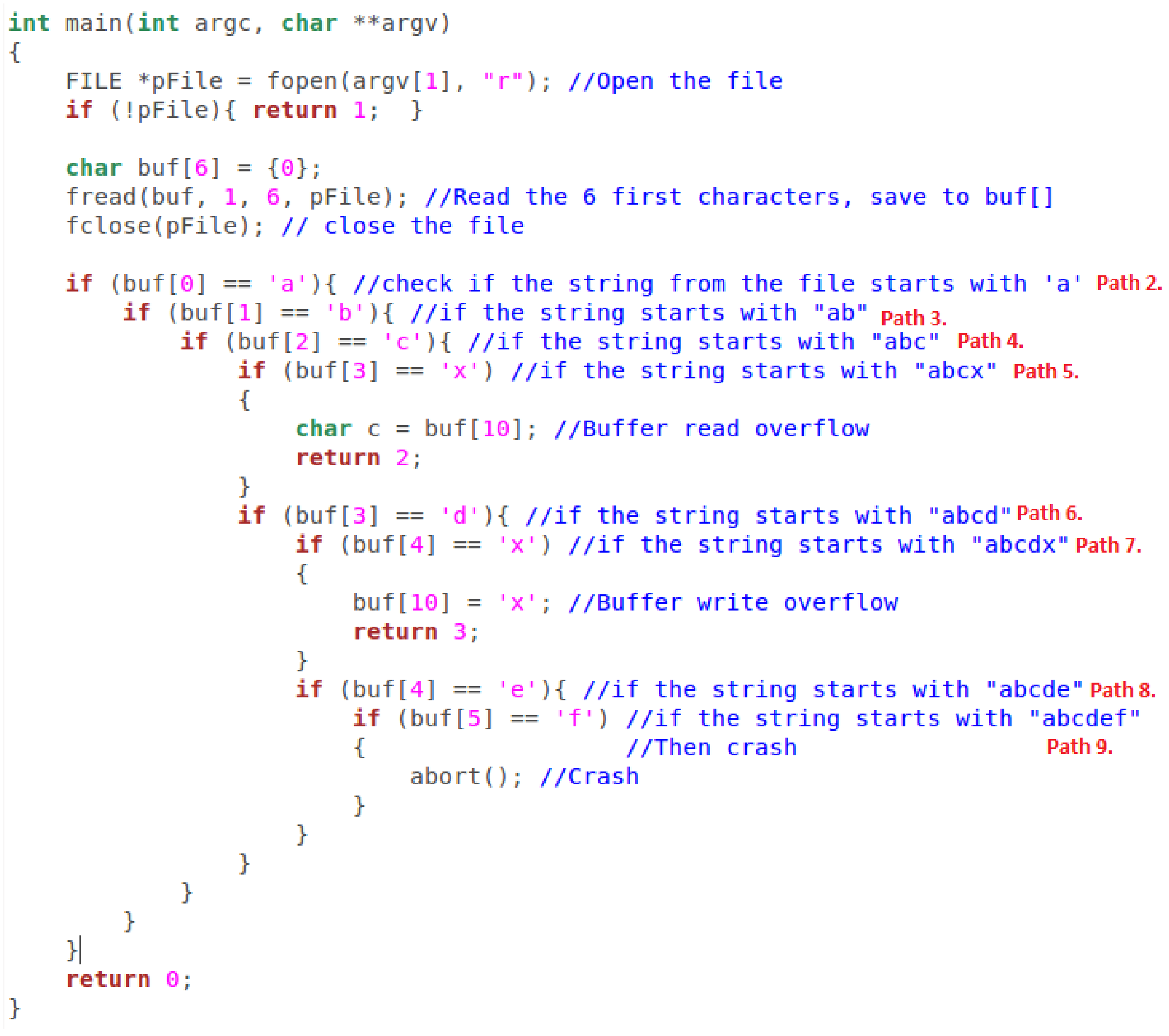
The first thing to notice is the process timing box. This gives information about the total time the fuzzer has been running, and time since last new path, unique crash, or unique hang, was found. A new path means the fuzzer has found a new path in the program code. A unique crash implies a crash has been found on a code path that has not crashed in a previous case. A unique hang is essentially the same as a unique crash, except a hang means the program was unresponsive for more than a set amount of time without properly crashing.

Next thing to notice is the overall results box. It informs about total test cycles done, total paths, unique crashes, and unique hangs. Total test cycles mean the total amount of times AFL has cycled through all its test on all paths found. The test program is minimal and fast; therefore, it has a cycle count of 10.1k (10 100). On a complex system, the first cycle could take more than a day. The number is now displayed in green since AFL considers itself done and probably will not find any new things. Total paths are the total number of paths that AFL has located in the program code. Unique crashes and hangs are the total number of crashes or hangs that have been found on a unique code path.

The rest of the status screen contains more in-depth information about the fuzzing session. There are mainly two variables essential for the average user. Execution speed, showing the number of times the program is re-executed per second; a higher number means faster fuzzing. Lastly total crashes; the total amounts of tests that have made the program crash, and how many of those are considered unique. A high number of non-unique crashes means many different test cases crash in the same execution-path. Only the unique crashes, and hangs, are saved in the output folder of AFL.

5.3 Initial testing

To evaluate how a fuzzer can be used, and get a clearer picture of fuzzing, a small test program, called fuzzTester, was made. In Figure 2, below, is the source code for fuzzTester.c together with comments trying to explain the code.



```
int main(int argc, char **argv)
{
    FILE *pFile = fopen(argv[1], "r"); //Open the file
    if (!pFile){ return 1;  }

    char buf[6] = {0};
    fread(buf, 1, 6, pFile); //Read the 6 first characters, save to buf[]
    fclose(pFile); // close the file

    if (buf[0] == 'a'){ //check if the string from the file starts with 'a' Path 2.
        if (buf[1] == 'b'){ //if the string starts with "ab" Path 3.
            if (buf[2] == 'c'){ //if the string starts with "abc" Path 4.
                if (buf[3] == 'x') //if the string starts with "abcx" Path 5.
                {
                    char c = buf[10]; //Buffer read overflow
                    return 2;
                }
                if (buf[3] == 'd'){ //if the string starts with "abcd" Path 6.
                    if (buf[4] == 'x') //if the string starts with "abcdx" Path 7.
                    {
                        buf[10] = 'x'; //Buffer write overflow
                        return 3;
                    }
                    if (buf[4] == 'e'){ //if the string starts with "abcde" Path 8.
                        if (buf[5] == 'f') //if the string starts with "abcdef"
                        {                   //Then crash                    Path 9.
                            abort(); //Crash
                        }
                    }
                }
            }
        }
    }
    return 0;
}
```

Figure 2. The source code of the test program, fuzzTester

The test program, fuzzTester was made to be used with a file format fuzzer. Simulating a program reading data from a text file, provided via the command line. The program contains two buffer overflows and one simulated crash, aborting the program execution. The program consists of several nested if statements, to make it a bit challenging for a fuzzer. For coverage-guided fuzzers, like AFL, the number of different paths that can be taken thru a program is essential. Therefore, path numbers are added in red color to illustrate that if the previous criteria is met, then path number x will be triggered.

The program reads a file containing a string of characters, saving the first 6 characters to a buffer. The string is then tested, one character at a time. If the first character is correct, the program will proceed to check the next character, and so on. If a character in the string does not match the check, then the program will end successfully. If the string is “abcdef”, it will result in a crash. If the string starts with “abcx” or “abcdx” it will result in a buffer overflow, but not necessarily a crash. The goal of the program is to test how a fuzzer can find the correct string, “abcdef”, to make the program crash and thereby find the simulated bug, and also to see how the buffer overflows are handled.

The starting point for a fuzzer, fuzzing this program, should be an empty file. Then the fuzzer has to mutate the whole file to make progress in the test program. Every new if-statement, and thereby code path, only requires the fuzzer to hit one specific character right. This makes for an easy target for a coverage-guided fuzzer, continuously finding new paths to start from. For a black-box fuzzer, without guidance or any good starting test case, the final crash would be impossible to find. To find the final crash, it would require the fuzzer to, randomly, make up a string starting with “abcdef”. A black-box fuzzer would likely only trigger the most shallow code paths (paths 2, 3 and 4 in Figure 2.).

5.3.1 Black-box

For the first test of the application, a black-box approach was used. In Figure 3 is a flowchart of black-box testing of the program fuzzTester. The Figure illustrates how the fuzzer starts with adding files from the input folder, into the queue. In the illustration, two input cases are added. One empty test case, and one longer, “better”, closer to the goal to make it a bit easier for the fuzzer. Each time the SUT is executed, the fuzzer chooses one of the test cases in the queue to mutate (to fuzz). A few examples of what the fuzzed file might end up like is shown in the green boxes. The fuzzed file is then sent to the SUT, which is monitored to see if it ends successfully or in a crash. Since this is a black-box test, without guidance, the queue folder will not get updated with any new tests, other than those provided from the start.

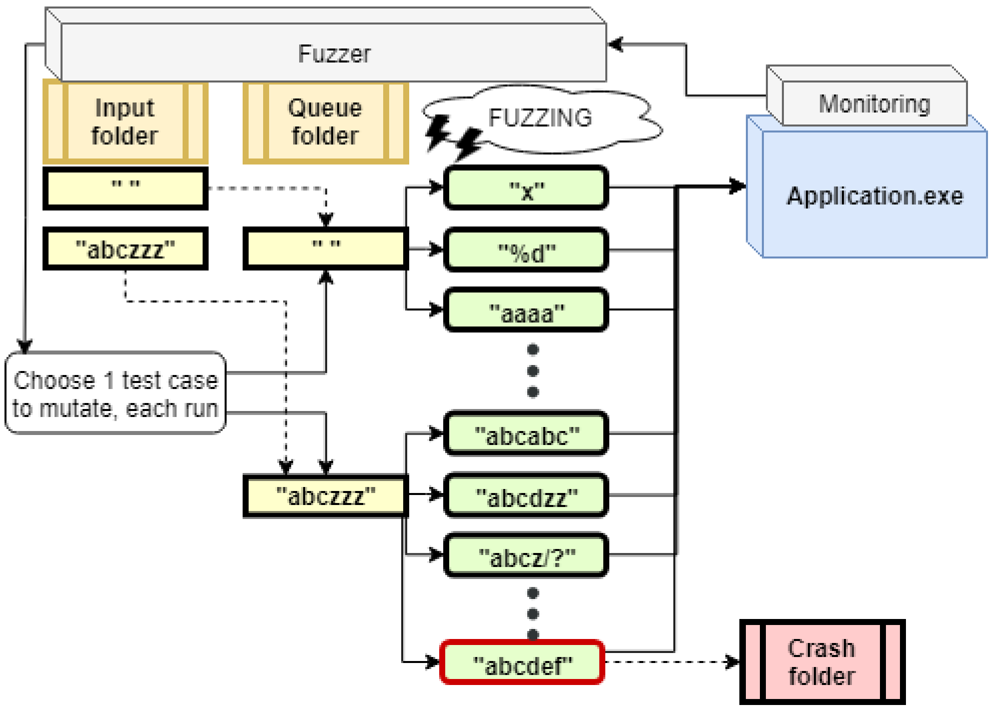
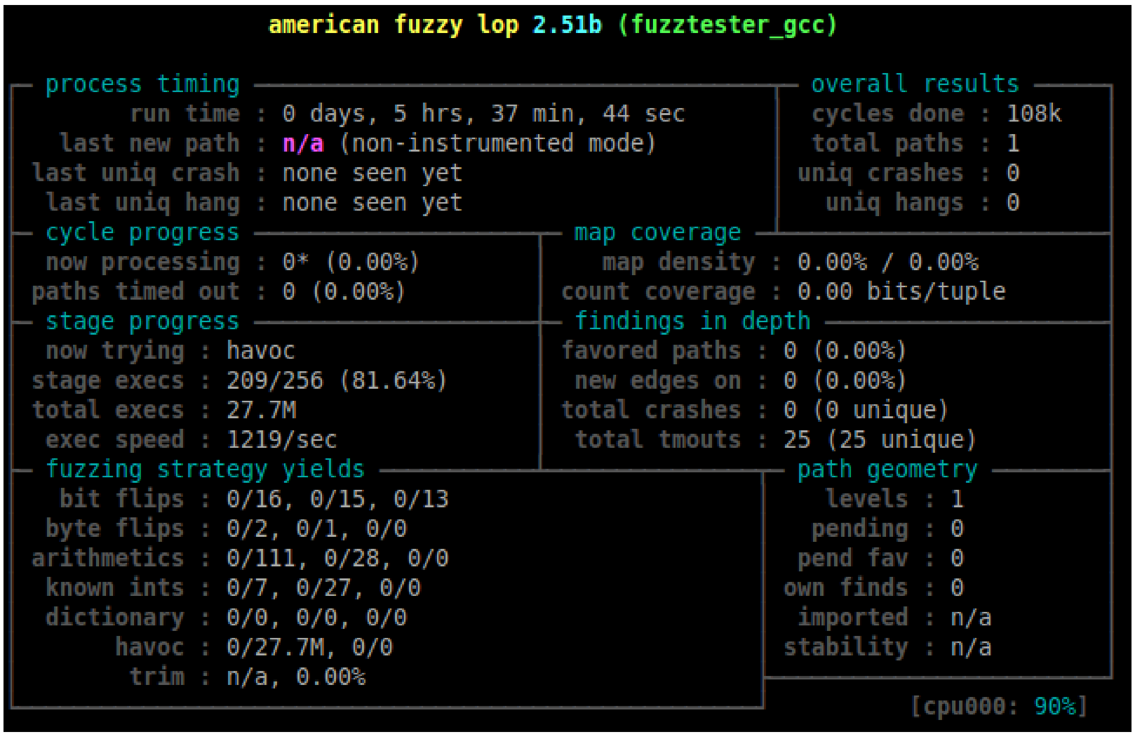


Figure 3. A basic illustration of black-box fuzzing the fuzzTester application, with AFL

In the first black-box test, only one empty file was used as input to AFL. The following command was used to start the fuzzing. The “-n” is added to inform AFL that it should run in non-instrumented mode, also known as black-box.

```
afl-fuzz -i ./i/ -o ./AFL_tests/gcc_standard/ -n ./fuzztester_gcc @@
```



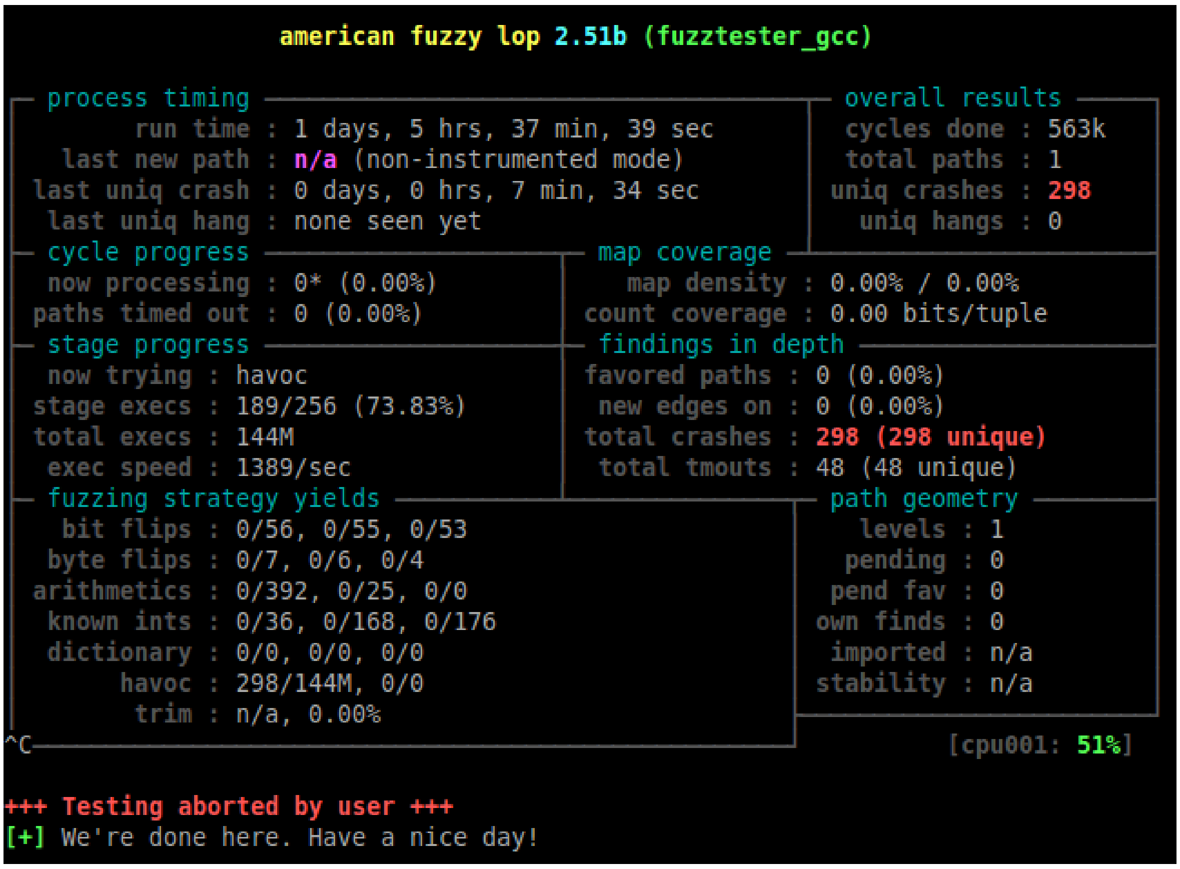
```
                american fuzzy lop 2.51b (fuzztester_gcc)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 5 hrs, 37 min, 44 sec      │  cycles done : 108k   │
│   last new path : n/a (non-instrumented mode)         │  total paths : 1      │
│ last uniq crash : none seen yet                       │ uniq crashes : 0      │
│  last uniq hang : none seen yet                       │   uniq hangs : 0      │
├─ cycle progress ────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 0* (0.00%)    │    map density : 0.00% / 0.00%          │
│ paths timed out : 0 (0.00%)     │ count coverage : 0.00 bits/tuple        │
├─ stage progress ────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc             │ favored paths : 0 (0.00%)               │
│ stage execs : 209/256 (81.64%)  │  new edges on : 0 (0.00%)               │
│ total execs : 27.7M             │ total crashes : 0 (0 unique)            │
│  exec speed : 1219/sec          │  total tmouts : 25 (25 unique)          │
├─ fuzzing strategy yields ───────┴───────────────┬─ path geometry ────────┤
│   bit flips : 0/16, 0/15, 0/13                   │    levels : 1          │
│  byte flips : 0/2, 0/1, 0/0                      │   pending : 0          │
│ arithmetics : 0/111, 0/28, 0/0                   │  pend fav : 0          │
│  known ints : 0/7, 0/27, 0/0                     │ own finds : 0          │
│  dictionary : 0/0, 0/0, 0/0                      │  imported : n/a        │
│       havoc : 0/27.7M, 0/0                       │ stability : n/a        │
│        trim : n/a, 0.00%                         ├────────────────────────┘
└──────────────────────────────────────────────────┘          [cpu000: 90%]
```

Figure 4. Black-box fuzzing of fuzzTester

The fuzzer was run for five hours, but as expected from this test, nothing interesting was found.

For the next test, the same black-box scenario as previous was used. Except for this time, instead of using only an empty file as input test case, a better file was used; “abczzz”. This will give the fuzzer a head-start, deeper into the fuzzTester program.

```
afl-fuzz -i ./i/ -o ./AFL_tests/gcc_standard_good_test_case/ -n ./fuzztester_gcc @@
```



```
                  american fuzzy lop 2.51b (fuzztester_gcc)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 1 days, 5 hrs, 37 min, 39 sec      │  cycles done : 563k   │
│   last new path : n/a (non-instrumented mode)        │  total paths : 1      │
│ last uniq crash : 0 days, 0 hrs, 7 min, 34 sec       │ uniq crashes : 298    │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 0* (0.00%)        │    map density : 0.00% / 0.00%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 0.00 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 0 (0.00%)             │
│ stage execs : 189/256 (73.83%)      │  new edges on : 0 (0.00%)             │
│ total execs : 144M                  │ total crashes : 298 (298 unique)      │
│  exec speed : 1389/sec              │  total tmouts : 48 (48 unique)        │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 0/56, 0/55, 0/53                      │    levels : 1          │
│  byte flips : 0/7, 0/6, 0/4                         │   pending : 0          │
│ arithmetics : 0/392, 0/25, 0/0                      │  pend fav : 0          │
│  known ints : 0/36, 0/168, 0/176                    │ own finds : 0          │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 298/144M, 0/0                         │ stability : n/a        │
│        trim : n/a, 0.00%                            ├────────────────────────┘
^C────────────────────────────────────────────────────┘          [cpu001: 51%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
```

Figure 5. Black-box fuzzing of fuzzTester, with a good starting test case

This time the results were more interesting. The fuzzer kept finding crashing inputs, every few minutes. The fuzzer was run for more than one day to get a good sample of test results.

By looking through all the 298 cases, it was found out that every crashing input starts with "abcdx" which means they crash at the write overflow. Many of the cases were identical, most commonly "abcdxz"; which shows that the fuzzer does not automatically remove identical test results. This also shows that the fuzzer can easily mutate two correct characters; mutates "abczzz" into "abcdxz". Very close to the final goal, "abcdef", but still it was not found. One case did stand out from the rest, a bunch of unknown characters making the program crash; "扡揰꙰砆成扢扢扢扢扢扢扢扢扢扢扢扢扢拿扢扢扢扢扢扢扢扢扢b".

The test was later run for one more day, without any new results.

5.3.2 White-box

For the following tests, a white-box fuzzing approach is used. In Figure 6 is a flow-chart of white-box testing of the test program. The basics are the same as with the black-box approach shown in Figure 3. The difference here is that instrumentations are used to give AFL's monitoring part more information of the internal execution of the test application. With that information, the fuzzer knows when a new path gets triggered. When a new path is triggered, the queue folder in automatically updated with the test case that triggered that new path. These new test cases can then be used as starting points for coming mutations.

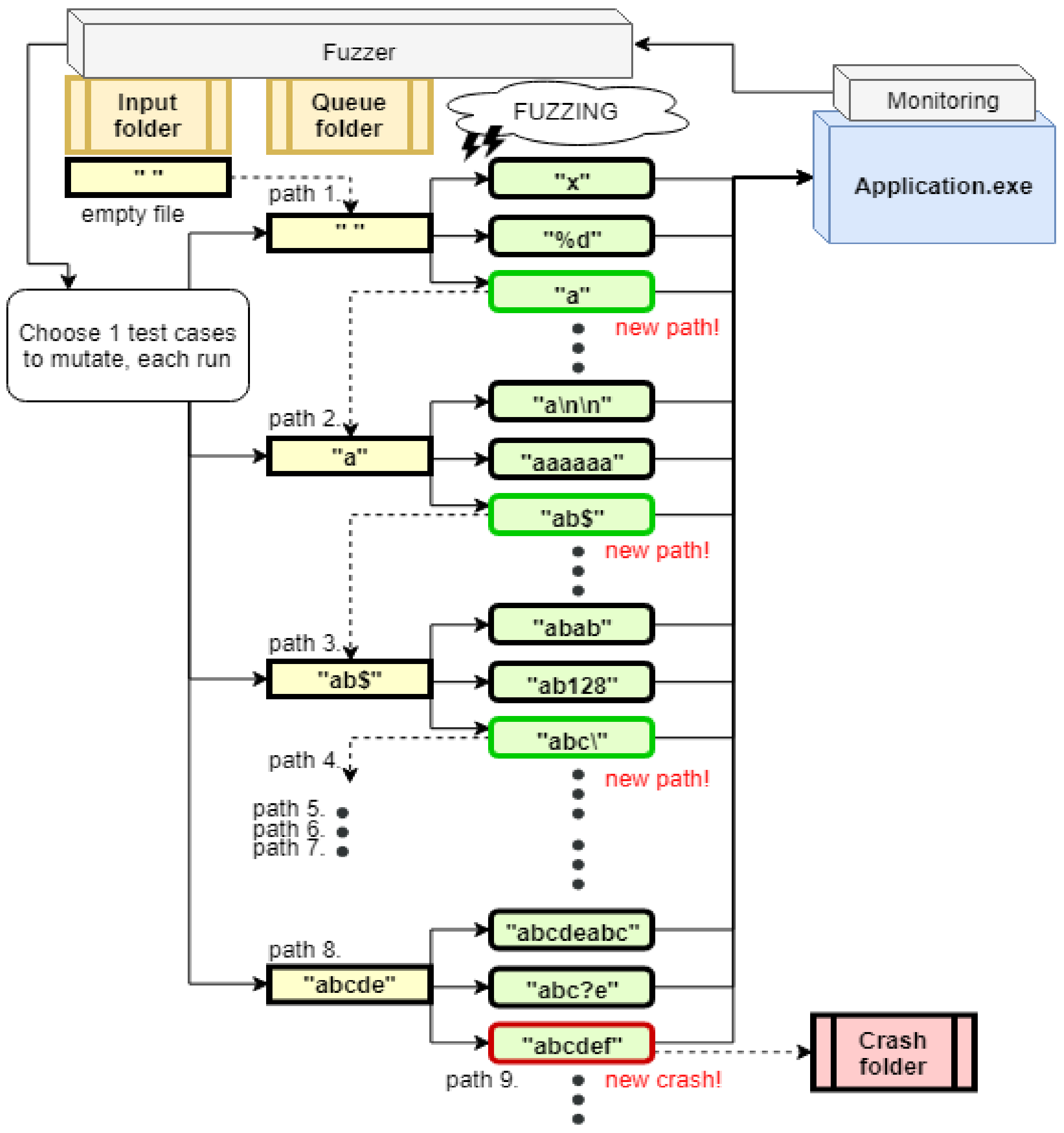


Figure 6. A basic illustration of how the test program is fuzzed by AFL, with the help of instrumentations

For testing AFL was used and the target, fuzzTester.c, was compiled with afl-gcc in a few different ways. Afl-gcc is the same as the standard gcc compiler plus binary instrumentations. Afl-gcc uses environment variables for compilation options, which can be set right before the compilation. According to the AFL-documentations [20], afl-gcc uses strict optimizations by default. For my simple test program, AFL_DONT_OPTIMIZE=1 has to be set to ignore all optimizations. My test program is too simple so otherwise vital parts of it would be optimized, ignored.

For some parts of the test, address sanitizer will be used to help the fuzzer to detect buffer overflows. Address sanitizer is a tool that, during compilation, adds instrumentations that will detect if any memory corruption happens. When using address sanitizer, AFL_USE_ASAN=1 must be set. Also, Because of the way address sanitizer uses memory, the target must be compiled in 32bit mode, therefore -m32 is added [20]. All tests were run as single instances, on one processor core.

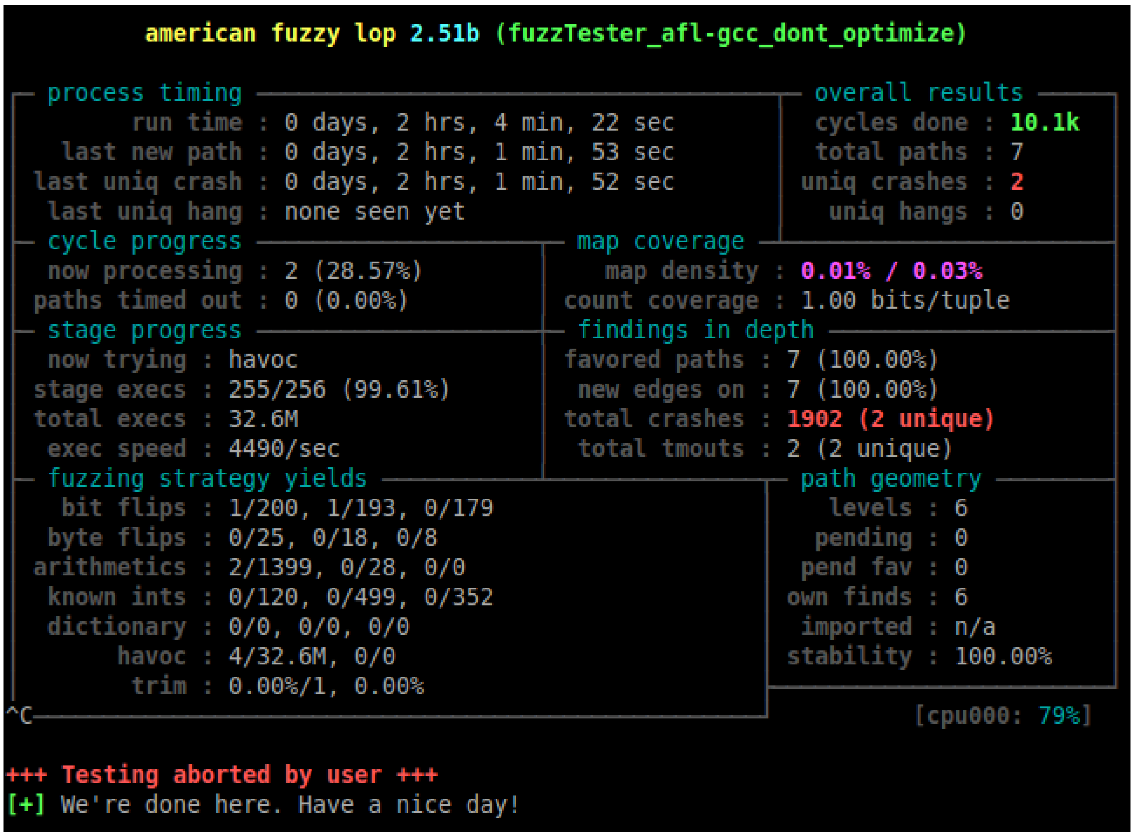
For the first test, the program fuzzTester.c was compiled with standard settings, non-optimized, 64-bit.

Command used to compile: env AFL_DONT_OPTIMIZE=1 afl-gcc fuzzTester.c

Program was named: fuzzTester_afl-gcc_dont_optimize

Command used to run the fuzzer: afl-fuzz -i ./i/ -o ./AFL_tests/afl-gcc_dont_optimize/ ./fuzzTester_afl-gcc_dont_optimize @@

The command is the standard for running a single fuzzing instance on a target reading from a file. The fuzzed file gets inserted in place of the @@. The directory ./i/ should contain good test cases for the SUT. In this case, it contained only one empty file, which is good enough for this small test.



```
          american fuzzy lop 2.51b (fuzzTester_afl-gcc_dont_optimize)

┌─ process timing ─────────────────────────────────────┬─ overall results ─┐
│        run time : 0 days, 2 hrs, 4 min, 22 sec        │  cycles done : 10.1k │
│   last new path : 0 days, 2 hrs, 1 min, 53 sec        │  total paths : 7     │
│ last uniq crash : 0 days, 2 hrs, 1 min, 52 sec        │ uniq crashes : 2     │
│  last uniq hang : none seen yet                       │   uniq hangs : 0     │
├─ cycle progress ──────────────────┬─ map coverage ─┴───────────────────┤
│  now processing : 2 (28.57%)      │    map density : 0.01% / 0.03%      │
│ paths timed out : 0 (0.00%)       │ count coverage : 1.00 bits/tuple    │
├─ stage progress ──────────────────┼─ findings in depth ─────────────────┤
│  now trying : havoc               │ favored paths : 7 (100.00%)         │
│ stage execs : 255/256 (99.61%)    │  new edges on : 7 (100.00%)         │
│ total execs : 32.6M               │ total crashes : 1902 (2 unique)     │
│  exec speed : 4490/sec            │  total tmouts : 2 (2 unique)        │
├─ fuzzing strategy yields ─────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 1/200, 1/193, 0/179                  │    levels : 6          │
│  byte flips : 0/25, 0/18, 0/8                      │   pending : 0          │
│ arithmetics : 2/1399, 0/28, 0/0                    │  pend fav : 0          │
│  known ints : 0/120, 0/499, 0/352                  │ own finds : 6          │
│  dictionary : 0/0, 0/0, 0/0                        │  imported : n/a        │
│       havoc : 4/32.6M, 0/0                         │ stability : 100.00%    │
│        trim : 0.00%/1, 0.00%                       ├────────────────────────┘
^C────────────────────────────────────────────────────┘          [cpu000: 79%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
```

Figure 7. White-box fuzzing of fuzzTester, 64-bit.

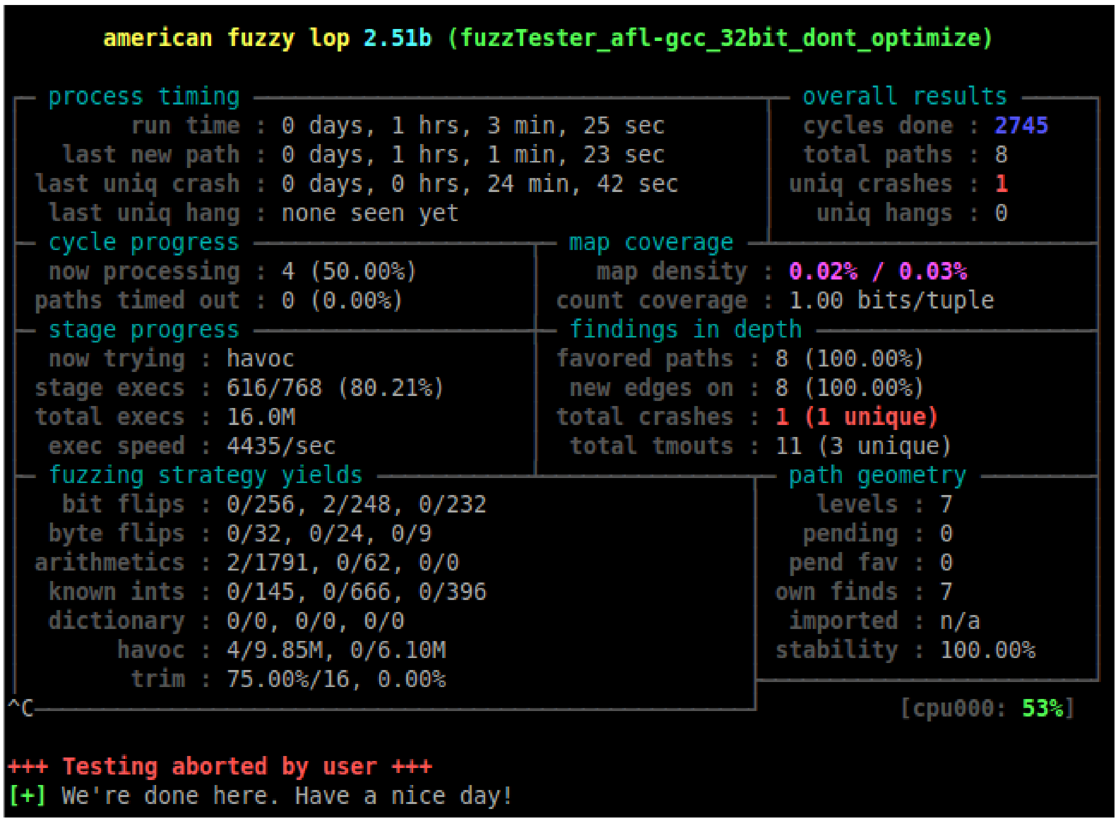
The fuzzer was run for 2 hours, although no new paths or crashes were found past the first 3 minutes. The crashes found were on; “abcdef” the final abort, and “abcdx” the write overflow. It did not crash on the read overflow, “abcx”.

Most embedded systems are compiled in 32bit rather than 64bit. The following test is the same as above, except it was compiled in 32bit instead of 64bit.

Command used to compile: `env AFL_DONT_OPTIMIZE=1 afl-gcc fuzzTester.c -m32`

Program was named: fuzzTester_afl-gcc_32bit_dont_optimize

Command used to run the fuzzer: `afl-fuzz -i ./i/ -o ./AFL_tests/afl-gcc_32bit_dont_optimize/ ./fuzzTester_afl-gcc_32bit_dont_optimize @@`



```
      american fuzzy lop 2.51b (fuzzTester_afl-gcc_32bit_dont_optimize)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 1 hrs, 3 min, 25 sec       │  cycles done : 2745  │
│   last new path : 0 days, 1 hrs, 1 min, 23 sec       │  total paths : 8     │
│ last uniq crash : 0 days, 0 hrs, 24 min, 42 sec      │ uniq crashes : 1     │
│  last uniq hang : none seen yet                      │   uniq hangs : 0     │
├─ cycle progress ────────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 4 (50.00%)        │    map density : 0.02% / 0.03%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.00 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : havoc                 │ favored paths : 8 (100.00%)           │
│ stage execs : 616/768 (80.21%)      │  new edges on : 8 (100.00%)           │
│ total execs : 16.0M                 │ total crashes : 1 (1 unique)          │
│  exec speed : 4435/sec              │  total tmouts : 11 (3 unique)         │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 0/256, 2/248, 0/232                   │    levels : 7          │
│  byte flips : 0/32, 0/24, 0/9                       │   pending : 0          │
│ arithmetics : 2/1791, 0/62, 0/0                     │  pend fav : 0          │
│  known ints : 0/145, 0/666, 0/396                   │ own finds : 7          │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 4/9.85M, 0/6.10M                      │ stability : 100.00%    │
│        trim : 75.00%/16, 0.00%                      ├────────────────────────┘
^C────────────────────────────────────────────────────┘          [cpu000: 53%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
```

Figure 8. White-box fuzzing of fuzzTester, 32-bit.

This time the fuzzer only found the final crash, not the buffer write overflow which was found in 64bit mode. It did traverse the code paths containing the overflows, that can be confirmed by looking in the queue folder in the AFL output directory. The folder contains all possible test cases as different files containing the text: “”, “a”, “ab”, “abc”, “abcd”, “abcx”, “abcdx”, “abcde”, and the crash folder containing the crashing case “abcdef”.

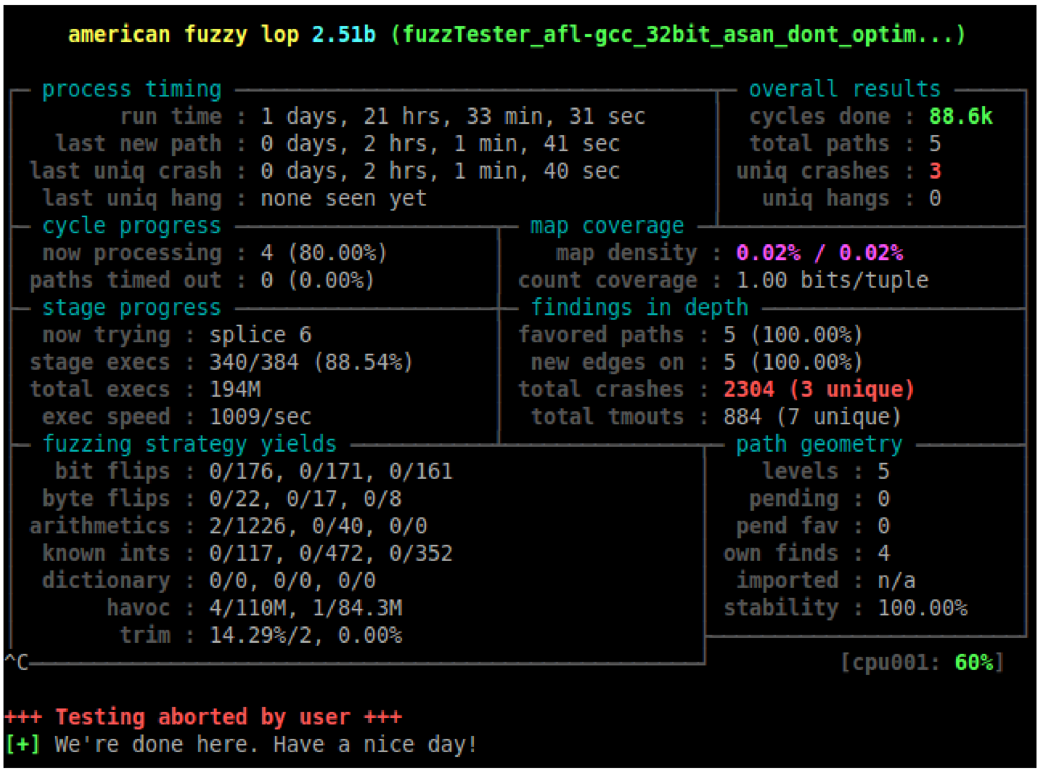
For the final test, address sanitizer was used.

Command used to compile: `env AFL_USE_ASAN=1 AFL_DONT_OPTIMIZE=1 afl-gcc fuzzTester.c -fsanitize=address -m32`

Program was named: fuzzTester_afl-gcc_32bit_asan_dont_optimize

Command used to run the fuzzer: `afl-fuzz -i ./i/ -o ./AFL_tests/afl-gcc_32bit_asan_dont_optimize/ -m 600 ./fuzzTester_afl-gcc_32bit_asan_dont_optimize @@`

When using address sanitizer, a lot of virtual memory is allocated. Therefore -m 600 needs to be set to increase the memory limit for the process to 600mb (any less and it will crash before it even starts).



```
     american fuzzy lop 2.51b (fuzzTester_afl-gcc_32bit_asan_dont_optim...)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 1 days, 21 hrs, 33 min, 31 sec      │  cycles done : 88.6k  │
│   last new path : 0 days, 2 hrs, 1 min, 41 sec        │  total paths : 5      │
│ last uniq crash : 0 days, 2 hrs, 1 min, 40 sec        │ uniq crashes : 3      │
│  last uniq hang : none seen yet                       │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 4 (80.00%)        │    map density : 0.02% / 0.02%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.00 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : splice 6              │ favored paths : 5 (100.00%)            │
│ stage execs : 340/384 (88.54%)      │  new edges on : 5 (100.00%)            │
│ total execs : 194M                  │ total crashes : 2304 (3 unique)        │
│  exec speed : 1009/sec              │  total tmouts : 884 (7 unique)         │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 0/176, 0/171, 0/161                    │    levels : 5          │
│  byte flips : 0/22, 0/17, 0/8                        │   pending : 0          │
│ arithmetics : 2/1226, 0/40, 0/0                      │  pend fav : 0          │
│  known ints : 0/117, 0/472, 0/352                    │ own finds : 4          │
│  dictionary : 0/0, 0/0, 0/0                          │  imported : n/a        │
│       havoc : 4/110M, 1/84.3M                        │ stability : 100.00%    │
│        trim : 14.29%/2, 0.00%                        ├────────────────────────┘
^C────────────────────────────────────────────────────┘          [cpu001: 60%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
```

Figure 9. White-box fuzzing of fuzzTester, 32-bit, address sanitizer activated

This time the fuzzer was able to find all the crashes, as expected when using address sanitizer. The downside of using address sanitizer was that the execution speed was significantly decreased, from 4000 executions per second down to 1000. To gain additional insights the tool afl-plot was used to create a plot of when the crashes were found. The plot shows that the first crash, “abcx”, was found fast while there was a long time between the two last findings, requiring close to two days to find all crashes.

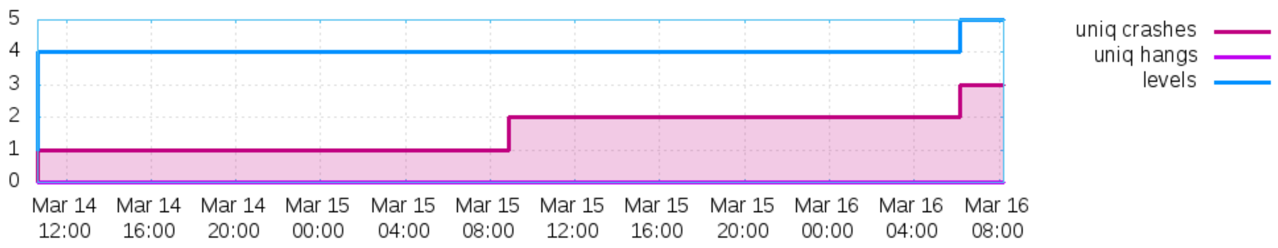


Figure 10. A plot of crashes over time

5.4 Testing on internal software

The software chosen for testing is a command line program, parsing XML files, parameter files, and command line parameters. The program outputs a compressed binary representation of the input file and parameters. This type of file can then provide the parameters to the embedded system. From here on the program tested will be referred to as the system under test, SUT.

The reasons why it was chosen are; it is a good target for a local fuzzer like AFL, it is a simple and small program so that focus can be put on actually fuzzing rather than setting up the system to be able to fuzz. The SUT is used in internal development only and since may contain more easy bugs to find.

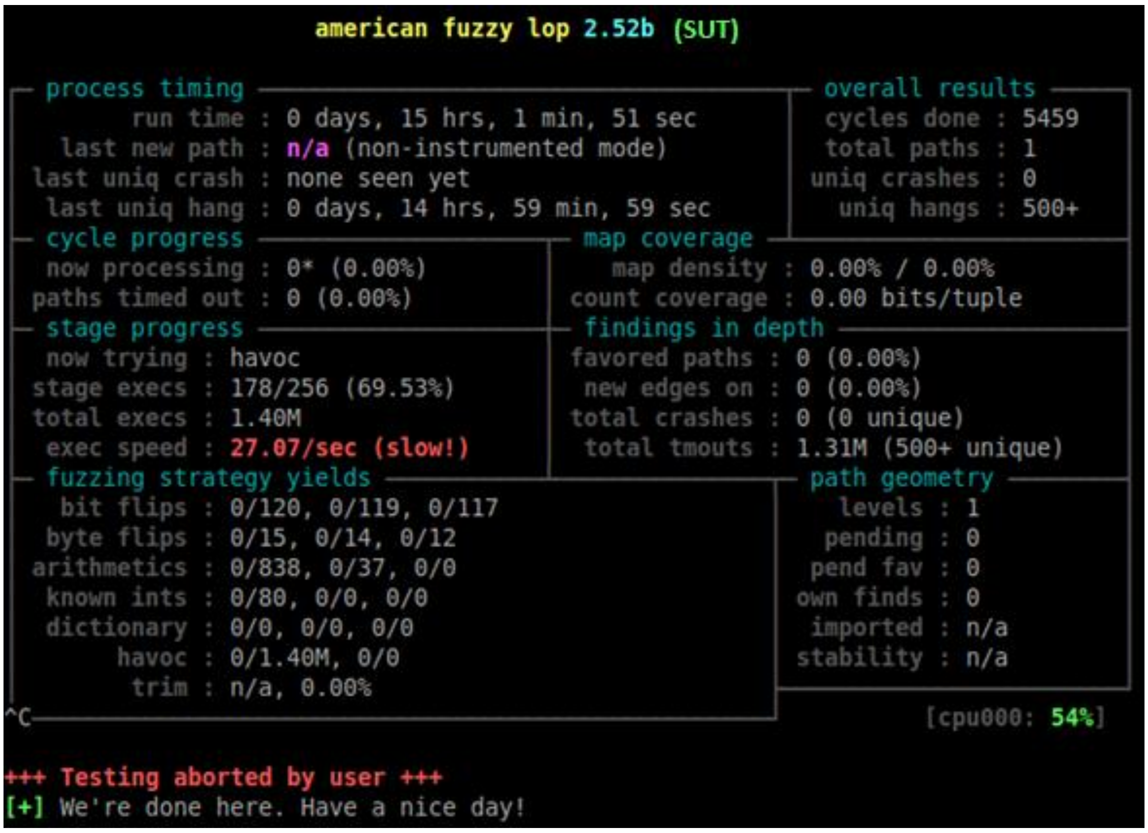
There are at least three layers that can be targeted in the SUT; parsing of XML files, parameter files, or command line parameters. For this test, the parameter file was chosen as the target input. Those files are small and simple, basically some lines of parameters. That makes for an excellent fuzzing target. Compared to an XML file which is relatively big with texts that need to be correctly formatted, that makes for a more difficult target for a fuzzer. Command line parameters could also be a target, but bugs found in a command line is not as exciting as those found in a file, and AFL only supports command line fuzzing on an experimental stage.

5.4.1 Black-box

For the first test, a black-box approach was used. As input test case, only one short valid test file was used. The file, containing the parameter-string, “100663296 11 0”, was put into the input folder.

The fuzzing was started with the command: `afl-fuzz -i ./i/ -o ./o/ -n ./path/to/SUT [SUT options] @@`

The “-n” parameter informs AFL that it should run in non-instrumented mode or black box mode. The @@ will be automatically replaced with a fuzzed file.



```
                      american fuzzy lop 2.52b (SUT)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 15 hrs, 1 min, 51 sec      │  cycles done : 5459  │
│   last new path : n/a (non-instrumented mode)        │  total paths : 1     │
│ last uniq crash : none seen yet                      │ uniq crashes : 0     │
│  last uniq hang : 0 days, 14 hrs, 59 min, 59 sec     │   uniq hangs : 500+  │
├─ cycle progress ────────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 0* (0.00%)        │    map density : 0.00% / 0.00%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 0.00 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : havoc                 │ favored paths : 0 (0.00%)             │
│ stage execs : 178/256 (69.53%)      │  new edges on : 0 (0.00%)             │
│ total execs : 1.40M                 │ total crashes : 0 (0 unique)          │
│  exec speed : 27.07/sec (slow!)     │  total tmouts : 1.31M (500+ unique)   │
├─ fuzzing strategy yields ───────────┴───────────┬─ path geometry ───────────┤
│   bit flips : 0/120, 0/119, 0/117               │    levels : 1             │
│  byte flips : 0/15, 0/14, 0/12                  │   pending : 0             │
│ arithmetics : 0/838, 0/37, 0/0                  │  pend fav : 0             │
│  known ints : 0/80, 0/0, 0/0                    │ own finds : 0             │
│  dictionary : 0/0, 0/0, 0/0                     │  imported : n/a           │
│       havoc : 0/1.40M, 0/0                      │ stability : n/a           │
│        trim : n/a, 0.00%                        ├───────────────────────────┘
^C────────────────────────────────────────────────┘             [cpu000: 54%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
```

Figure 11. Black-box fuzzing of the SUT

The first thing to notice here is that the last new path is not available when doing a black box test. That means the fuzzer is blind to what paths are taken in the program. Also, the execution speed is very slow, probably due to the huge amount of timeouts, ”total tmouts” in Figure 11. In this test, the program is considered timeout if the execution of the program takes more than 50ms.

No crashes were found, but many hangs. A hang occurs when the program times out for much longer than 50ms, about 1s or more. The high amount of ”unique” hangs found are due to the fuzzer being blind, fuzzing a black-box. Many of them can be identical cases or similar. The test cases producing hangs are saved in the output directory in a folder named hangs. “./o/hangs/”

By manually running the SUT program with a few test files from the hangs folder, they were confirmed to indeed hang the program. The cause of the hangs could be evaluated, by quickly looking through all the files in the hangs folder. It appears that the program correctly refuses files that are not the correct length and format since there is no really deformed test case. But the program hangs if there is a non-numerical character, in a correct format.

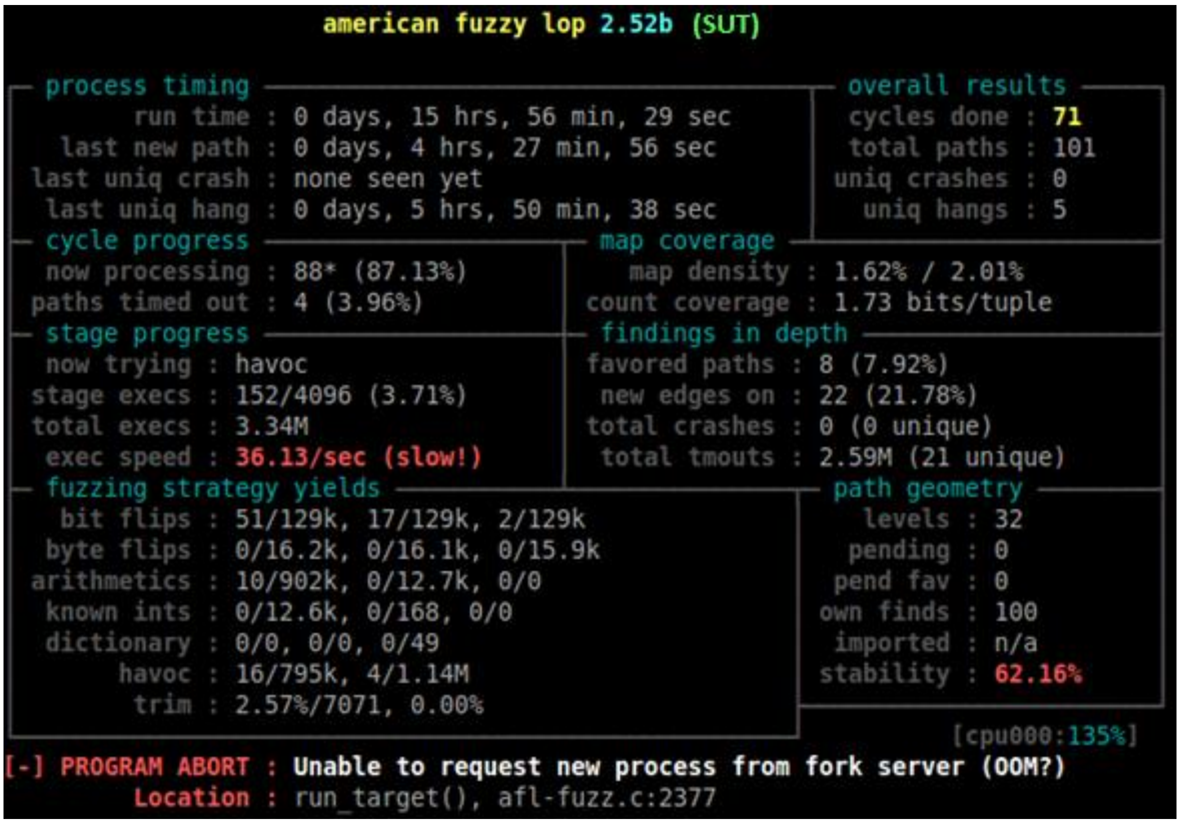
Samples that will produce a hang; 1?0663296 11 0 : 1006N3296 11 0 : 100663296 11/0

5.4.2 White-box

For the next test, the SUT was built with AFL instrumentations to make a white-box approach to fuzzing. The C compiler was set to afl-gcc, and the C++ compiler was set to afl-g++.

The command used to launch the fuzzer is the same as in the previous test, except this time without the -n parameter. Instrumentations inserted by AFL during the building of the program is used.

The fuzzing was started with the command: afl-fuzz -i ./i/ -o ./o/ ./path/to/SUT [SUT options] @@



```
                    american fuzzy lop 2.52b (SUT)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 15 hrs, 56 min, 29 sec      │  cycles done : 71     │
│   last new path : 0 days, 4 hrs, 27 min, 56 sec       │  total paths : 101    │
│ last uniq crash : none seen yet                       │ uniq crashes : 0      │
│  last uniq hang : 0 days, 5 hrs, 50 min, 38 sec       │   uniq hangs : 5      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 88* (87.13%)      │    map density : 1.62% / 2.01%         │
│ paths timed out : 4 (3.96%)         │ count coverage : 1.73 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 8 (7.92%)              │
│ stage execs : 152/4096 (3.71%)      │  new edges on : 22 (21.78%)            │
│ total execs : 3.34M                 │ total crashes : 0 (0 unique)           │
│  exec speed : 36.13/sec (slow!)     │  total tmouts : 2.59M (21 unique)      │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 51/129k, 17/129k, 2/129k              │    levels : 32         │
│  byte flips : 0/16.2k, 0/16.1k, 0/15.9k             │   pending : 0          │
│ arithmetics : 10/902k, 0/12.7k, 0/0                 │  pend fav : 0          │
│  known ints : 0/12.6k, 0/168, 0/0                   │ own finds : 100        │
│  dictionary : 0/0, 0/0, 0/49                        │  imported : n/a        │
│       havoc : 16/795k, 4/1.14M                      │ stability : 62.16%     │
│        trim : 2.57%/7071, 0.00%                     ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu000:135%]
[-] PROGRAM ABORT : Unable to request new process from fork server (OOM?)
         Location : run_target(), afl-fuzz.c:2377
```

Figure 12. White-box fuzzing of the SUT

As can be seen in the bottom of Figure 12, the fuzzer aborted the process after running over 15h, because it was out of memory (OOM). This could be a side effect of fuzzing. Running a process millions of times with manipulated inputs could result in memory leaks. But that is not a problem, the results so far can be analyzed, and the fuzzer can quickly be restarted from where it was stopped.

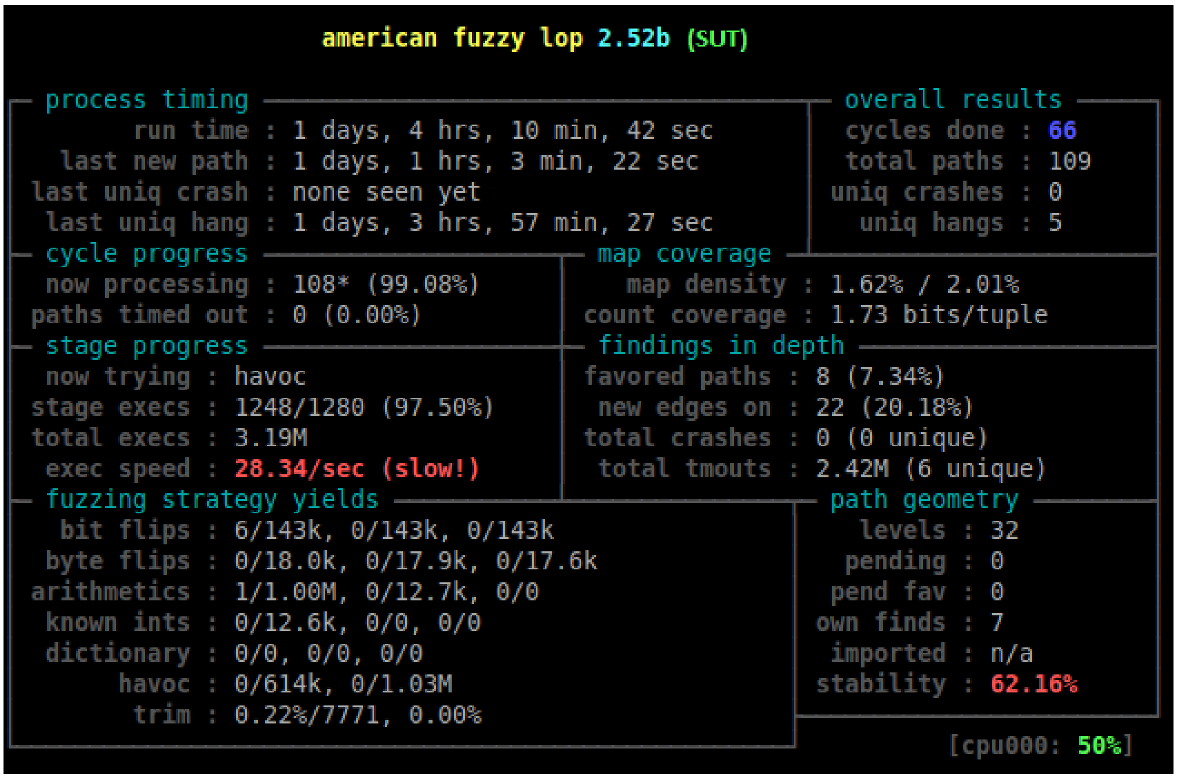
By looking at the status screen, this approach does have the same problems as the earlier black-box test. The execution speed is slow, and most tests result in timeouts. But this time, with the help of white-box instrumentations, the fuzzer can track the paths taken in the program. Total 101 paths have been found, and 5 unique hangs, no crashes.

Hangs found: °00663296 11 0 : 000663296 11 0\8A : 000663296 11 0 :

100661 066+2   6 12+2\B9 : ±00663294 11 0

The results seem to be the same as with the black-box test, except this time there were only 5 unique hangs, instead of 500. Making the results easier to interpret.

One unexpected result here is the stability value, which can be seen in red in Figure 12's lower right corner. Per the AFL documentation [20], the stability shows the consistency of observed traces. The stability should be at 100% when the system behaves in the same way for the exact same input. If it is lower than 100%, that means that the SUT in some cases behaves differently although the same input is used. A small drop in the stability value can be normal, but a bigger drop, as in this case shown in red can be a problem for the fuzzers performance. There can be several reasons to a low stability value; some functionality can be designed to behave randomly, the program might try to manipulate leftover temporary files or memory, or the application might use uninitialized memory that somehow gains random values.



```
                  american fuzzy lop 2.52b (SUT)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 1 days, 4 hrs, 10 min, 42 sec       │  cycles done : 66     │
│   last new path : 1 days, 1 hrs, 3 min, 22 sec        │  total paths : 109    │
│ last uniq crash : none seen yet                       │ uniq crashes : 0      │
│  last uniq hang : 1 days, 3 hrs, 57 min, 27 sec       │   uniq hangs : 5      │
├─ cycle progress ────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 108* (99.08%) │    map density : 1.62% / 2.01%         │
│ paths timed out : 0 (0.00%)     │ count coverage : 1.73 bits/tuple       │
├─ stage progress ────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc             │ favored paths : 8 (7.34%)              │
│ stage execs : 1248/1280 (97.50%)│  new edges on : 22 (20.18%)            │
│ total execs : 3.19M             │ total crashes : 0 (0 unique)           │
│  exec speed : 28.34/sec (slow!) │  total tmouts : 2.42M (6 unique)       │
├─ fuzzing strategy yields ───────┴───────────────┬─ path geometry ────────┤
│   bit flips : 6/143k, 0/143k, 0/143k            │    levels : 32         │
│  byte flips : 0/18.0k, 0/17.9k, 0/17.6k         │   pending : 0          │
│ arithmetics : 1/1.00M, 0/12.7k, 0/0             │  pend fav : 0          │
│  known ints : 0/12.6k, 0/0, 0/0                 │ own finds : 7          │
│  dictionary : 0/0, 0/0, 0/0                     │  imported : n/a        │
│       havoc : 0/614k, 0/1.03M                   │ stability : 62.16%     │
│        trim : 0.22%/7771, 0.00%                 ├────────────────────────┘
└─────────────────────────────────────────────────┘          [cpu000: 50%]
```

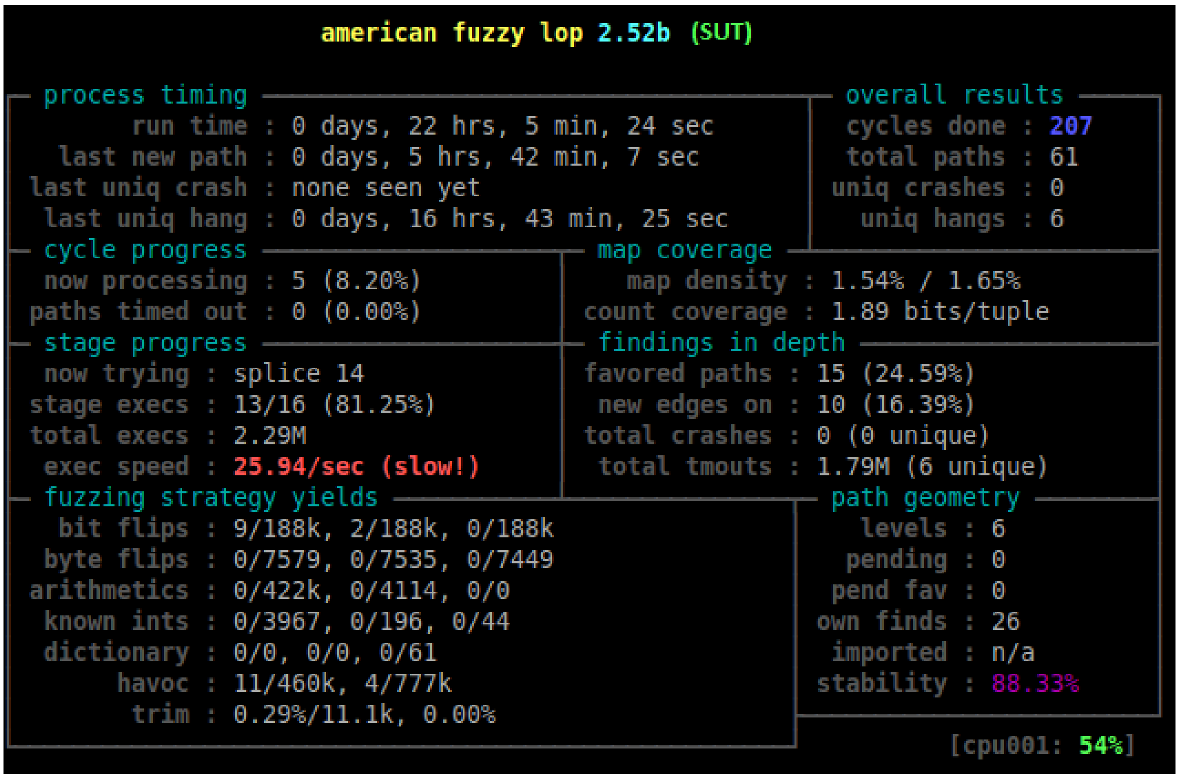
Figure 13. Continuation of the previous test

The test was continued for one more day, without any new results.

Next step was to do a test with address sanitizer activated. That should detect if there is any memory corruption. The SUT was compiled as before, but with address sanitizer on. Before starting the fuzzer, the test cases produced by the previous fuzzing round was run through an AFL tool, afl-cmin, to minimize the test cases deleting redundant tests. 101 tests were minimized to 36. These 36 test cases were put into the input folder, to give a good starting point for the next fuzzing session.

The test is started with the same command line as before, except now with -m 700. Address sanitizer requires a lot of memory; therefore, the memory limit of the process is increased to 700mb.

```
afl-fuzz -i ./i/ -o ./o/ -m 700 ./path/to/SUT [SUT options] @@
```

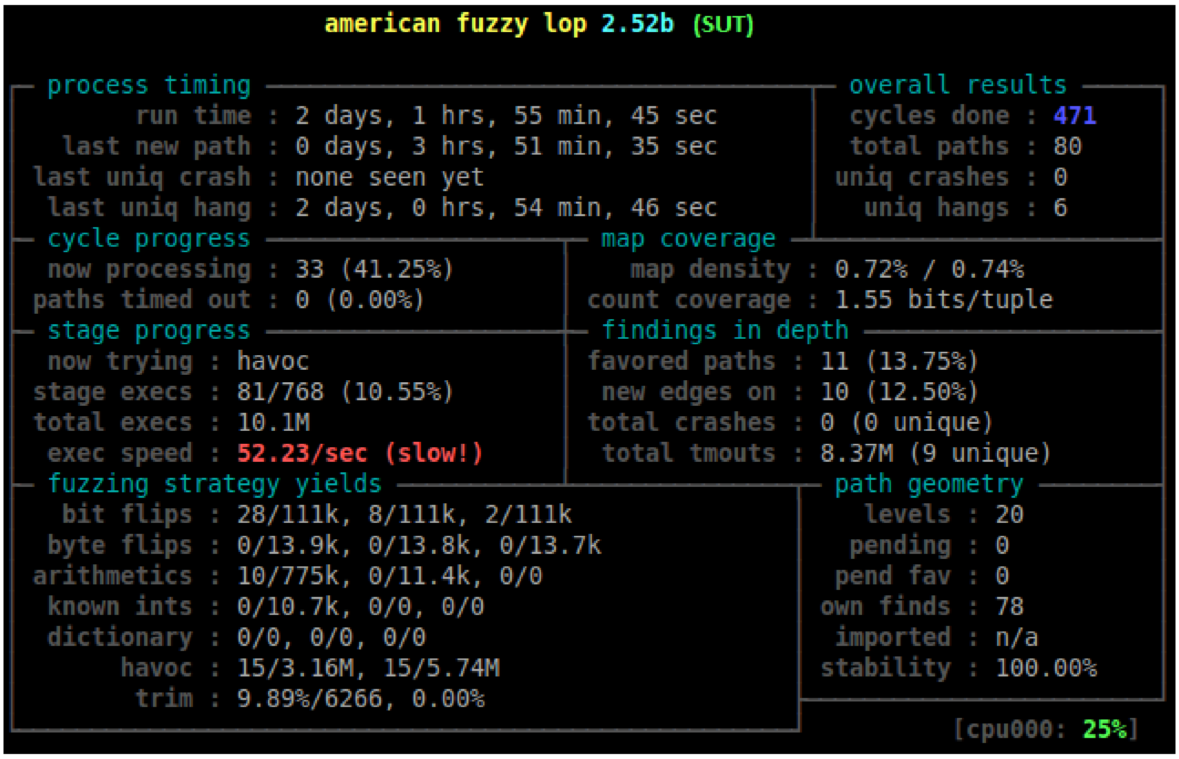


```
                     american fuzzy lop 2.52b (SUT)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 22 hrs, 5 min, 24 sec      │  cycles done : 207    │
│   last new path : 0 days, 5 hrs, 42 min, 7 sec       │  total paths : 61     │
│ last uniq crash : none seen yet                      │ uniq crashes : 0      │
│  last uniq hang : 0 days, 16 hrs, 43 min, 25 sec     │   uniq hangs : 6      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 5 (8.20%)         │    map density : 1.54% / 1.65%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.89 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : splice 14             │ favored paths : 15 (24.59%)           │
│ stage execs : 13/16 (81.25%)        │  new edges on : 10 (16.39%)           │
│ total execs : 2.29M                 │ total crashes : 0 (0 unique)          │
│  exec speed : 25.94/sec (slow!)     │  total tmouts : 1.79M (6 unique)      │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 9/188k, 2/188k, 0/188k                 │    levels : 6          │
│  byte flips : 0/7579, 0/7535, 0/7449                 │   pending : 0          │
│ arithmetics : 0/422k, 0/4114, 0/0                    │  pend fav : 0          │
│  known ints : 0/3967, 0/196, 0/44                    │ own finds : 26         │
│  dictionary : 0/0, 0/0, 0/61                         │  imported : n/a        │
│       havoc : 11/460k, 4/777k                        │ stability : 88.33%     │
│        trim : 0.29%/11.1k, 0.00%                     ├────────────────────────┘
└──────────────────────────────────────────────────────┘          [cpu001: 54%]
```

Figure 14. Fuzzing of the SUT, compiled with instrumentations from AFL and address sanitizer

Using address sanitizer did not yield any different results than previous tests. The same few hangs, all with the same characteristics. Interestingly though the stability value was better.

Lastly, some changes were made to the source code, to disable parts of the program, to get a more fuzz-friendly target. The components that were disabled were; CRC checks, reading of the XML file, and writing of the output file. The program should now contain fewer obstacles for the fuzzer to go over, leaving more time for the tested functionality, parsing of the parameter file. This time the fuzzer was run without address sanitizer.



```
                    american fuzzy lop 2.52b (SUT)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 2 days, 1 hrs, 55 min, 45 sec      │  cycles done : 471   │
│   last new path : 0 days, 3 hrs, 51 min, 35 sec      │  total paths : 80    │
│ last uniq crash : none seen yet                      │ uniq crashes : 0     │
│  last uniq hang : 2 days, 0 hrs, 54 min, 46 sec      │   uniq hangs : 6     │
├─ cycle progress ────────────────┬─ map coverage ─────┴──────────────────────┤
│  now processing : 33 (41.25%)   │    map density : 0.72% / 0.74%           │
│ paths timed out : 0 (0.00%)     │ count coverage : 1.55 bits/tuple         │
├─ stage progress ────────────────┼─ findings in depth ───────────────────────┤
│  now trying : havoc             │ favored paths : 11 (13.75%)              │
│ stage execs : 81/768 (10.55%)   │  new edges on : 10 (12.50%)              │
│ total execs : 10.1M             │ total crashes : 0 (0 unique)             │
│  exec speed : 52.23/sec (slow!) │  total tmouts : 8.37M (9 unique)         │
├─ fuzzing strategy yields ───────┴───────────────┬─ path geometry ──────────┤
│   bit flips : 28/111k, 8/111k, 2/111k           │    levels : 20           │
│  byte flips : 0/13.9k, 0/13.8k, 0/13.7k         │   pending : 0            │
│ arithmetics : 10/775k, 0/11.4k, 0/0             │  pend fav : 0            │
│  known ints : 0/10.7k, 0/0, 0/0                 │ own finds : 78           │
│  dictionary : 0/0, 0/0, 0/0                     │  imported : n/a          │
│       havoc : 15/3.16M, 15/5.74M                │ stability : 100.00%      │
│        trim : 9.89%/6266, 0.00%                 ├──────────────────────────┘
└─────────────────────────────────────────────────┘          [cpu000: 25%]
```

Figure 15. Fuzzing of a more fuzz-friendly version of the SUT

The results from this fuzz-friendly SUT were generally no different from previous tests. The same types of hangs were found. The change is that the execution speed is doubled, but still quite slow and with the same number of timeouts. A more significant change can be seen in the stability value, which is now at 100%. Apparently, some of the disabled features did link to the stability issue, but that did not seem to change the outcome.

6 Results

The tests were successful in giving a picture of how a fuzzer can be used, and what kind of problems that can occur. The initial tests showing how fuzzing should behave in a good scenario, while the later tests show that it is not as easy with a real software. When fuzzing the internal software, SUT, no crashes were found, but a few ways to make the SUT hang were discovered. Although finding crashes is the primary objective of fuzzing, hangs are equally important to detect to maintain a stable program. The way the hangs were triggered was trivial, which confirms what fuzzing is best at; catching the low hanging fruit.

6.1 Black- or white-box

White-box fuzzing should uncover more in-depth bugs in the SUT, but in this test, it did not, although it did show its strength on the initial small test program. The problems found in the SUT were shallow bugs that could be found easily by both a black- and white-box test. White box did provide more information and a smaller and better filtered batch of test results. That means there is less work with interpreting the results.

6.2 Address sanitizer

Although address sanitizer did prove useful in the initial small test program, it did not yield any further results when used on the SUT. Using address sanitizer significantly slows down the fuzzing process, as shown by the initial testing, increasing the time it takes to make a full fuzzing of the system. Other than time consumption, there is no known downside to using address sanitizer. So, if memory corruptions, like buffer overflows, are suspected or if time is not a problem, address sanitizer should be used.

6.3 Chosen method

AFL was well suited as a test method on the selected test application. No reason could be found as to how any other tool would have performed better, on this implementation. In the tests, the compilers gcc and g++ were used. If the compilers were replaced by the clang compiler, new options would be available. Especially LibFuzzer, or Honggfuzz which are more focused on clang.

6.4 Further testing

To get a more conclusive result, more testing would need to be done. Either other systems could be tested, or more approaches could be tested on the same SUT. In these tests the SUT was only tested through the parameters file, tests could also be made on the XML file. To get a more efficient fuzzing a fuzz-friendly build was tested, more testing could be made that way by gaining more knowledge about the inner workings of the SUT. With good knowledge about the SUT, strategical changes could be made in the source code to create a better fuzz-friendly build of the SUT. A fuzz-friendly build would strive for a system without unnecessary checks, file writing, or other time-consuming tasks while keeping the core functionality that is being tested intact.

All tests were done as single tasks, running on a single CPU core. Parallelization can quickly be done with a fuzzer, to utilize all available CPU cores. That would increase the fuzzing speed, to gain results faster.

If fuzzing were to be taken into use for further software testing, the developers would be in a key position to spot core functionality, suitable for fuzzing. Core functions could then be fuzzed throughout the development, to find at least the easy bugs quickly. Even an easy bug could cause significant problems if it is not found during development.

7 Conclusion

The results of the tests show that fuzzing can be a promising tool for testing software on an embedded system. On a well-tested system, it can be challenging to find new bugs. Since fuzzing is best at finding easy bugs, fuzzing would be best when used from the beginning of a project, using a fuzzer to find low hanging fruit quickly.

Fuzzing is a versatile tool for software testing. Fuzzing can be targeted to almost any part of a system, or piece of the code if there is enough knowledge about the SUT. This makes fuzzing into a great tool to have knowledge about, to be able to implement it when given the right situation.

This thesis will be used as a reference to what fuzzing is, and how it can be used. Fuzzing will be considered for future use in the department and compared to other testing methods. It can also be used as a reference point in the future, to see how fuzzing has changed over time.

8 Discussion

There are many sides of fuzzing. Fuzzing can be quick and easy to implement, or it may require lots of pre-work before starting. Fuzzing may find some bugs, but maybe not all. When fuzzing, and not finding bugs, it is difficult to know if the system actually does not contain bugs. The fuzzer might merely not be targeting the right spot, or set up in the right way, to find the bug.

Fuzzing as a technique is a moving subject. There are new open source fuzzers appearing, often improving some specific part of the fuzzing. But many of those lack the commitment to keep them updated, and thus they get outdated, or the new technique gets merged and used in a bigger project.

What I have learned from this is that even though fuzzing is a small relatively unknown subject of a to me unknown subject, software testing, there were plenty of rabbit holes to jump down into and get lost. When reading about new implementations of the subject, it was easy to lose track of the scope of this thesis. To get work done, prioritizations need to be done to keep the focus on what is essential.

While writing this thesis, a lot of time was put on understanding the basics. Also learning how to target different systems, and how to set up the system to be able to fuzz, took more time than expected. With that knowledge, further testing could focus more on testing different fuzzing approaches in varying situations.

For further reading on the subject, I would like to have the chance to read the book “Fuzzing for Software Security Testing and Quality Assurance, 2nd Edition” Released February 2018. The release was too late for it to be used for this thesis. This book is significantly newer than previous books on the subject, covering more modern fuzzing methods as evolutionary fuzzing and AFL.

References

[1] Wärtsilä, "This is Wärtsilä," [Online]. Available: https://www.wartsila.com/about. [Accessed 16 January 2018].

[2] P. Amini, A. Greene and M. Sutton, Fuzzing: Brute Force Vulnerability Discovery, Addison-Wesley Professional, 2007.

[3] D. Lebrero, "The tragedy of 100% code coverage," IG, 18 May 2016. [Online]. Available: http://labs.ig.com/code-coverage-100-percent-tragedy. [Accessed 17 March 2018].

[4] A. Kettunen, P. Pietikäinen, M. Laakso, A. Kauppi, J. Kuorilehto, E. Kurimo, M. Nyman, R. Kumpulainen and A. Kirichenko, "Fuzz testing: Beginner's guide," 21 September 2017. [Online]. Available: https://github.com/ouspg/fuzz-testing-beginners-guide. [Accessed 10 February 2018].

[5] A. Takanen, J. D. Demott and C. Miller, Fuzzing for Software Security Testing and Quality Assurance, Artech House, 2008.

[6] B. Miller, "Foreword for Fuzz Testing Book," The University of Wisconsin Madison, April 2008. [Online]. Available: http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html. [Accessed 17 March 2018].

[7] B. Perry, "Fuzzing workflows; a fuzz job from start to finish," Foxglove security, 15 March 2016. [Online]. Available: https://foxglovesecurity.com/2016/03/15/fuzzing-workflows-a-fuzz-job-from-start-to-finish/. [Accessed 17 March 2018].

[8] M. Hillman, "15 minute guide to fuzzing," MWR InfoSecurity, 8 August 2013. [Online]. Available: https://www.mwrinfosecurity.com/our-thinking/15-minute-guide-to-fuzzing/. [Accessed 10 January 2018].

[9] C. Miller and Z. N. J. Peterson, "Analysis of Mutation and Generation-Based Fuzzing," Independent Security Evaluators, 1 March 2007. [Online]. Available: https://www.defcon.org/images/defcon-15/dc15-presentations/Miller/Whitepaper/dc-15-miller-WP.pdf. [Accessed 31 January 2018].

[10] J. D. DeMott, R. J. Enbody and W. F. Punch, "Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing," 2007. [Online]. Available: https://www.blackhat.com/presentations/bh-usa-07/DeMott_Enbody_and_Punch/Whitepaper/bh-usa-07-demott_enbody_and_punch-WP.pdf. [Accessed 4 February 2018].

[11] Y. Zhang, "Evaluating Software Security Aspects through Fuzzing and Genetic Algorithms," 02 12 2008. [Online]. Available: http://resolver.tudelft.nl/uuid:8660f6f6-248a-4c50-8127-e8f8b3aab582. [Accessed 10 February 2018].

[12] M. Zalewski, "Pulling JPEGs out of thin air," lcamtuf's blog, 07 November 2014. [Online]. Available: https://lcamtuf.blogspot.fi/2014/11/pulling-jpegs-out-of-thin-air.html. [Accessed 03 February 2018].

[13] M. Zalewski, "AFL README," [Online]. Available: http://lcamtuf.coredump.cx/afl/README.txt. [Accessed 7 January 2018].

[14] M. Zalewski, "american fuzzy lop (2.52b)," [Online]. Available: http://lcamtuf.coredump.cx/afl/. [Accessed 7 January 2018].

[15] "Honggfuzz," [Online]. Available: https://github.com/google/honggfuzz. [Accessed 30 March 2018].

[16] LLVM Project, "libFuzzer – a library for coverage-guided fuzz testing," LLVM Project, 14 March 2018. [Online]. Available: https://llvm.org/docs/LibFuzzer.html. [Accessed 14 March 2018].

[17] Peachtech, "PEACH FUZZER," 2018. [Online]. Available: https://www.peach.tech/products/peach-fuzzer/. [Accessed 3 April 2018].

[18] Synopsys, Inc., "Defensics Fuzz Testing," 2018. [Online]. Available: https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html. [Accessed 3 April 2018].

[19] PortSwigger Ltd., "Burp Suit Editions," PortSwigger, 2018. [Online]. Available: https://portswigger.net/burp. [Accessed 11 March 2018].

[20] M. Zalewski, *AFL documentation, found in /afl_root_folder/docs/.*

Appendix

This is a code example of fuzzing, extracted from American Fuzzy Lop's afl-fuzz.c. AFL consists of many fuzzing stages, as seen in the list below. Due to the size of the code, only one stage, random havoc, is added here as a reference. Random havoc is an example of random brute-force mutation of an input file. The code has comments for every stage, with a short description of what is happening.

```
/* Fuzzing stages */

enum {
  /* 00 */ STAGE_FLIP1,
  /* 01 */ STAGE_FLIP2,
  /* 02 */ STAGE_FLIP4,
  /* 03 */ STAGE_FLIP8,
  /* 04 */ STAGE_FLIP16,
  /* 05 */ STAGE_FLIP32,
  /* 06 */ STAGE_ARITH8,
  /* 07 */ STAGE_ARITH16,
  /* 08 */ STAGE_ARITH32,
  /* 09 */ STAGE_INTEREST8,
  /* 10 */ STAGE_INTEREST16,
  /* 11 */ STAGE_INTEREST32,
  /* 12 */ STAGE_EXTRAS_UO,
  /* 13 */ STAGE_EXTRAS_UI,
  /* 14 */ STAGE_EXTRAS_AO,
  /* 15 */ STAGE_HAVOC,
  /* 16 */ STAGE_SPLICE
};


  /****************
   * RANDOM HAVOC *
   ****************/

havoc_stage:

  stage_cur_byte = -1;

  /* The havoc stage mutation code is also invoked when splicing files;
if the
     splice_cycle variable is set, generate different descriptions and
such. */

  if (!splice_cycle) {

    stage_name  = "havoc";
    stage_short = "havoc";
    stage_max   = (doing_det ? HAVOC_CYCLES_INIT : HAVOC_CYCLES) *
                  perf_score / havoc_div / 100;

  } else {

    static u8 tmp[32];

    perf_score = orig_perf;

    sprintf(tmp, "splice %u", splice_cycle);
    stage_name  = tmp;
    stage_short = "splice";
    stage_max   = SPLICE_HAVOC * perf_score / havoc_div / 100;

  }

  if (stage_max < HAVOC_MIN) stage_max = HAVOC_MIN;

  temp_len = len;

  orig_hit_cnt = queued_paths + unique_crashes;

  havoc_queued = queued_paths;

  /* We essentially just do several thousand runs (depending on
perf_score)
     where we take the input file and make random stacked tweaks. */

  for (stage_cur = 0; stage_cur < stage_max; stage_cur++) {

    u32 use_stacking = 1 << (1 + UR(HAVOC_STACK_POW2));

    stage_cur_val = use_stacking;

    for (i = 0; i < use_stacking; i++) {

      switch (UR(15 + ((extras_cnt + a_extras_cnt) ? 2 : 0))) {

        case 0:

          /* Flip a single bit somewhere. Spooky! */

          FLIP_BIT(out_buf, UR(temp_len << 3));
          break;

        case 1:

          /* Set byte to interesting value. */

          out_buf[UR(temp_len)] =
interesting_8[UR(sizeof(interesting_8))];
          break;

        case 2:

          /* Set word to interesting value, randomly choosing endian. */

          if (temp_len < 2) break;

          if (UR(2)) {

            *(u16*)(out_buf + UR(temp_len - 1)) =
              interesting_16[UR(sizeof(interesting_16) >> 1)];

          } else {

            *(u16*)(out_buf + UR(temp_len - 1)) = SWAP16(
              interesting_16[UR(sizeof(interesting_16) >> 1)]);

          }

          break;

        case 3:
```

```c
          /* Set dword to interesting value, randomly choosing endian. */

          if (temp_len < 4) break;

          if (UR(2)) {

            *(u32*)(out_buf + UR(temp_len - 3)) =
              interesting_32[UR(sizeof(interesting_32) >> 2)];

          } else {

            *(u32*)(out_buf + UR(temp_len - 3)) = SWAP32(
              interesting_32[UR(sizeof(interesting_32) >> 2)]);

          }

          break;

        case 4:

          /* Randomly subtract from byte. */

          out_buf[UR(temp_len)] -= 1 + UR(ARITH_MAX);
          break;

        case 5:

          /* Randomly add to byte. */

          out_buf[UR(temp_len)] += 1 + UR(ARITH_MAX);
          break;

        case 6:

          /* Randomly subtract from word, random endian. */

          if (temp_len < 2) break;

          if (UR(2)) {

            u32 pos = UR(temp_len - 1);

            *(u16*)(out_buf + pos) -= 1 + UR(ARITH_MAX);

          } else {

            u32 pos = UR(temp_len - 1);
            u16 num = 1 + UR(ARITH_MAX);

            *(u16*)(out_buf + pos) =
              SWAP16(SWAP16(*(u16*)(out_buf + pos)) - num);

          }

          break;

        case 7:

          /* Randomly add to word, random endian. */

          if (temp_len < 2) break;

          if (UR(2)) {
```

```
          u32 pos = UR(temp_len - 1);

          *(u16*)(out_buf + pos) += 1 + UR(ARITH_MAX);

        } else {

          u32 pos = UR(temp_len - 1);
          u16 num = 1 + UR(ARITH_MAX);

          *(u16*)(out_buf + pos) =
            SWAP16(SWAP16(*(u16*)(out_buf + pos)) + num);

        }

        break;

      case 8:

        /* Randomly subtract from dword, random endian. */

        if (temp_len < 4) break;

        if (UR(2)) {

          u32 pos = UR(temp_len - 3);

          *(u32*)(out_buf + pos) -= 1 + UR(ARITH_MAX);

        } else {

          u32 pos = UR(temp_len - 3);
          u32 num = 1 + UR(ARITH_MAX);

          *(u32*)(out_buf + pos) =
            SWAP32(SWAP32(*(u32*)(out_buf + pos)) - num);

        }

        break;

      case 9:

        /* Randomly add to dword, random endian. */

        if (temp_len < 4) break;

        if (UR(2)) {

          u32 pos = UR(temp_len - 3);

          *(u32*)(out_buf + pos) += 1 + UR(ARITH_MAX);

        } else {

          u32 pos = UR(temp_len - 3);
          u32 num = 1 + UR(ARITH_MAX);

          *(u32*)(out_buf + pos) =
            SWAP32(SWAP32(*(u32*)(out_buf + pos)) + num);

        }

        break;
        case 10:

          /* Just set a random byte to a random value. Because,
             why not. We use XOR with 1-255 to eliminate the
             possibility of a no-op. */

          out_buf[UR(temp_len)] ^= 1 + UR(255);
          break;

        case 11 ... 12: {

            /* Delete bytes. We're making this a bit more likely
               than insertion (the next option) in hopes of keeping
               files reasonably small. */

            u32 del_from, del_len;

            if (temp_len < 2) break;

            /* Don't delete too much. */

            del_len = choose_block_len(temp_len - 1);

            del_from = UR(temp_len - del_len + 1);

            memmove(out_buf + del_from, out_buf + del_from + del_len,
                    temp_len - del_from - del_len);

            temp_len -= del_len;

            break;

          }

        case 13:

          if (temp_len + HAVOC_BLK_XL < MAX_FILE) {

            /* Clone bytes (75%) or insert a block of constant bytes
(25%). */

            u8  actually_clone = UR(4);
            u32 clone_from, clone_to, clone_len;
            u8* new_buf;

            if (actually_clone) {

              clone_len  = choose_block_len(temp_len);
              clone_from = UR(temp_len - clone_len + 1);

            } else {

              clone_len = choose_block_len(HAVOC_BLK_XL);
              clone_from = 0;

            }

            clone_to   = UR(temp_len);

            new_buf = ck_alloc_nozero(temp_len + clone_len);

            /* Head */

            memcpy(new_buf, out_buf, clone_to);
            /* Inserted part */

            if (actually_clone)
              memcpy(new_buf + clone_to, out_buf + clone_from,
clone_len);
            else
              memset(new_buf + clone_to,
                     UR(2) ? UR(256) : out_buf[UR(temp_len)], clone_len);

            /* Tail */
            memcpy(new_buf + clone_to + clone_len, out_buf + clone_to,
                   temp_len - clone_to);

            ck_free(out_buf);
            out_buf = new_buf;
            temp_len += clone_len;

          }

          break;

        case 14: {

            /* Overwrite bytes with a randomly selected chunk (75%) or
fixed
               bytes (25%). */

            u32 copy_from, copy_to, copy_len;

            if (temp_len < 2) break;

            copy_len  = choose_block_len(temp_len - 1);

            copy_from = UR(temp_len - copy_len + 1);
            copy_to   = UR(temp_len - copy_len + 1);

            if (UR(4)) {

              if (copy_from != copy_to)
                memmove(out_buf + copy_to, out_buf + copy_from,
copy_len);

            } else memset(out_buf + copy_to,
                          UR(2) ? UR(256) : out_buf[UR(temp_len)],
copy_len);

            break;

          }

        /* Values 15 and 16 can be selected only if there are any extras
           present in the dictionaries. */

        case 15: {

            /* Overwrite bytes with an extra. */

            if (!extras_cnt || (a_extras_cnt && UR(2))) {

              /* No user-specified extras or odds in our favor. Let's use
an
                 auto-detected one. */
              u32 use_extra = UR(a_extras_cnt);
              u32 extra_len = a_extras[use_extra].len;
              u32 insert_at;

              if (extra_len > temp_len) break;

              insert_at = UR(temp_len - extra_len + 1);
              memcpy(out_buf + insert_at, a_extras[use_extra].data,
extra_len);

            } else {

              /* No auto extras or odds in our favor. Use the dictionary.
*/

              u32 use_extra = UR(extras_cnt);
              u32 extra_len = extras[use_extra].len;
              u32 insert_at;

              if (extra_len > temp_len) break;

              insert_at = UR(temp_len - extra_len + 1);
              memcpy(out_buf + insert_at, extras[use_extra].data,
extra_len);

            }

            break;

          }

        case 16: {

            u32 use_extra, extra_len, insert_at = UR(temp_len + 1);
            u8* new_buf;

            /* Insert an extra. Do the same dice-rolling stuff as for the
               previous case. */

            if (!extras_cnt || (a_extras_cnt && UR(2))) {

              use_extra = UR(a_extras_cnt);
              extra_len = a_extras[use_extra].len;

              if (temp_len + extra_len >= MAX_FILE) break;

              new_buf = ck_alloc_nozero(temp_len + extra_len);

              /* Head */
              memcpy(new_buf, out_buf, insert_at);

              /* Inserted part */
              memcpy(new_buf + insert_at, a_extras[use_extra].data,
extra_len);

            } else {

              use_extra = UR(extras_cnt);
              extra_len = extras[use_extra].len;

              if (temp_len + extra_len >= MAX_FILE) break;

              new_buf = ck_alloc_nozero(temp_len + extra_len);
              /* Head */
              memcpy(new_buf, out_buf, insert_at);

              /* Inserted part */
              memcpy(new_buf + insert_at, extras[use_extra].data,
extra_len);

            }

            /* Tail */
            memcpy(new_buf + insert_at + extra_len, out_buf + insert_at,
                   temp_len - insert_at);

            ck_free(out_buf);
            out_buf   = new_buf;
            temp_len += extra_len;

            break;

          }

      }

    }

    if (common_fuzz_stuff(argv, out_buf, temp_len))
      goto abandon_entry;

    /* out_buf might have been mangled a bit, so let's restore it to its
       original size and shape. */

    if (temp_len < len) out_buf = ck_realloc(out_buf, len);
    temp_len = len;
    memcpy(out_buf, in_buf, len);

    /* If we're finding new stuff, let's run for a bit longer, limits
       permitting. */

    if (queued_paths != havoc_queued) {

      if (perf_score <= HAVOC_MAX_MULT * 100) {
        stage_max  *= 2;
        perf_score *= 2;
      }

      havoc_queued = queued_paths;

    }

  }
```