Master Thesis
Computing Science - Cyber Security

Radboud University

Fuzzing Using Side Channel Information

Author:
Stefan Popa
1027672

First supervisor/assessor:
Assoc. Prof. Erik Poll
erikpoll@cs.ru.nl

Start Date:
June 1, 2023
End Date:
February 26th, 2024

Second assessor:
Assoc. Prof. Stjepan Picek
stjepan.picek@ru.nl

Thesis Supervisors/Readers:
Cristian Daniele
cristian.daniele@ru.nl

Seyed Behnam Andarzian
seyedbehnam.andarzian@ru.nl

February 25, 2024

Abstract

Fuzzing or fuzz testing is a popular technique used to test software programs. The general idea is that an automated software creates and injects malformed input into a (software) system in order to detect any defects or vulnerabilities that might exist. In other words, this allows us to discover whether the system we are testing has any negative reactions to the malformed inputs, which might indicate security or quality issues. In most cases, the type of vulnerabilities that are being discovered by a fuzzer include buffer overflows, denial of service (DoS), cross-site scripting, and code injections.

Many improvements have been made to fuzz testing over time, such as expanding the code coverage of a fuzzer by using an evaluation, or fitness, function to enhance the test case mutations. With this, it is ensured that the fuzzer will find most, if not all, above mentioned vulnerabilities within a program.

In this thesis, we evaluate the feasibility of introducing a new type of vulnerabilities into the scope of fuzz testing - side-channel vulnerabilities. Our goal is that, in case a side-channel vulnerability exists within the program that is being analyzed, a fuzzer can use this vulnerability to improve code coverage. Moreover, we also aim for the fuzzer to be able to extract the secret information using the side-channel vulnerability. For example, if the program being fuzzed checks character by character whether the password given as input is correct, our fuzzer could find this password by maximizing the execution time, because finding correct characters means that the program will take longer to execute. We will call the fuzzer which uses side-channel vulnerabilities during the fuzzing process $\text{SCFuzz}_{\text{LibAFL}}$.

Our original goal was to apply our fuzzer on software programs as well as on a hardware device, such as smart cards. However, due to the many difficulties we have encountered during the process of creating $\text{SCFuzz}_{\text{LibAFL}}$, we only present a detailed analysis of the fuzzer when applied on software programs which are vulnerable to a timing side-channel, and using only one of the possible approaches for integrating this side-channel during the fuzzing process. Taking these scope limitations into account, our conclusion is that our approach for using a timing side-channel during fuzzing can only be applied to a limited number of programs, and it is not always enough for code coverage or to discover the complete secret information.

Contents

Chapter 1

Introduction

Fuzzing, or fuzz testing, is an \an automated software testing method that injects invalid inputs into a system to reveal software defects and vulnerabilities. A fuzzing tool injects these inputs into the system and then monitors for exceptions such as crashes or information leakage" [1]. First proposed and developed by Miller et al. [13], fuzzing has become one of the more practical and popular ways to nd bugs in programs. It has been applied to various software, such as kernels or compilers, and it has a wide range of application areas, for example fault localization [19].

An extensive body of work exists in this area. Various survey papers [2, 11, 10] explore the current state-of-the-art in fuzz testing. Moreover, a wide range of new fuzzer implementations have been proposed and explored, using di erent techniques to improve on currently available solutions, such as using input-to-state correspondence to improve feedback-driven fuzzing, using machine learning to improve input fuzzing and exploiting memory usage to guide the fuzzer, amongst others [17, 1, 18, 6].

One method of improving fuzz testing that has not been explored as extensively is using side-channels to guide the fuzzing process. Side-channel attacks allow an adversary to uncover secret information using non-functional program behavior(s), called side-channels. Side-channels refer to timing information, power consumption or electromagnetic radiation, amongst others. Side-channels can have serious consequences, as for instance timing vulnerabilities have been found in Google's Keyczar Library [9] or how the more recent Meltdown and Spectre side-channel attacks [12, 8] allowed for critical vulnerabilities in modern processors to be exploited which resulted in the uncovering of secret information. Therefore, combining side-channels with fuzz testing could have interesting results.

[1] https://www.synopsys.com/glossary/what-is-fuzz-testing.html

DifFuzz [15] and QFuzz [16], as explained in more detail in Chapter 2 and Section 3.1.5, are the only two papers we could find on this topic. Their main focus is to use differential fuzzing for finding whether exploitable side-channel vulnerabilities exist. On the other hand, in our work we try to use side-channel information to to find the secret data, such as the correct password of a program, and eventually to guide the fuzzing process in terms of improving code coverage. If successful, this would be an important achievement, as this will allow for fuzzing to be used in cases where the source code is not available, or in cases where instrumentation is not an option. For example, when we only have access to the binaries or when we are trying to fuzz hardware devices, such as smart cards. Our original goal was to create a fully working fuzzer implementation that can be applied on smart cards, using different side-channels, however due to lack of time and issues we have encountered throughout our design process, we had to reduce our scope. Therefore, throughout this work we answer the following two main research question:

- *Can we use side-channel information about the execution time of a program to guide a fuzzer, such that it can replace the instrumentation in terms of code coverage? Moreover, can the same timing side-channel information allow the fuzzer to extract the secret data used by the program being fuzzed?*

Our contributions can be summarized as follows. In Chapter 2 we will introduce the main papers that have explored the possibility of combining side-channel information with fuzzing, as well as explain how these papers relate to our own work.

In Chapter 3 we will describe the main concepts and techniques we will be using throughout this thesis.

Chapter 4 will explain the design concept of adding timing side-channel information to a fuzzer - which we will call SCFuzz from this point onward. This explanation includes the required modifications, the various options in which a side-channel score can be implemented for a timing vulnerability and a high-level description of the steps taken by a fuzzer with timing side-channel information, as well as an overview of the attempts we have made at creating SCFuzz. Here we will also introduce a simple password checking program which is vulnerable to a timing side-channel, and a quick analysis of the results obtained by applying AFL on this program, which we will treat as a baseline for the results obtained with SCFuzz.

Chapter 5 details our initial attempts at creating SCFuzz, which involved modifying the fuzzers American Fuzzy Lop, or AFL and AFL++ to use timing side-channel information during the fuzzing process. We explain the main changes, the various versions and the main issues we believe to be the reasons why these attempts have failed.

In Chapter 6 we present our attempt at creating SCFuzz using LibAFL. We explain the different LibAFL modules that are necessary for a working fuzzer, our changes and additions to these modules for using timing side-channel information, the various versions and the reasoning behind them and an analysis of the results that led to our conclusion that our implementation of $\text{SCFuzz}_{\text{LibAFL}}$ can be used successfully only on a limited number of software programs.

In Chapter 7 we offer an overview of what we consider to be interesting remaining issues to tackle regarding the use of side-channels in fuzz testing.

Chapter 2

Related Work

There is a large body of work on fuzzers, increasing fuzzer e ciency and coverage and on using fuzzers to discover whether exploitable side-channel vulnerabilities exist. However, for the work we perform in our thesis, three papers are most relevant.

DifFuzz [15] uses di erential fuzzing, as explained in Section 3.1.2 and 3.1.5, to discover inputs that reveal side-channel attack vectors, with the main focus on time and space vulnerabilities. This is slightly di erent than our goal, e.g., use side-channel information to guide the fuzzer towards code coverage or nding the secret data. However, DifFuzz does o er a good basis on what changes are required to introduce side-channel information in the fuzzing process.

QFuzz [16] builds upon the above work. The main di erence is that it also allows for a quantitative measurement of the side channel vulnerabilities, or in other words, whether or not a detected vulnerability can, in fact, be exploited. It achieves this using a similar principle as DifFuzz, however it does not have any added value over DifFuzz for our own work. Therefore, we will mainly focus on and use DifFuzz.

Finally, in his thesis [14], Gerdriaan Mulder looked into guiding the AFL fuzzer as well as extending its coverage by using side-channel information on JavaCard applets. The main focus was on testing a modi ed version of AFL that used side-channel information on a password-checker, and guiding the fuzzing process using status words and timing information. However, the goal of the experiments is not clearly de ned and the conclusions that are being drawn are inconclusive.

Chapter 3

Technical Background

As described in Chapter 1, we will now give a detailed overview of the concepts and techniques we will be using throughout our work. In Section 3.1 we will o er a de nition of fuzzing and its related concepts, as well as describe the various techniques used within fuzz testing. Additionally, in 3.1.3, 3.1.4 and 3.1.5, the most important details about AFL, LibAFL and DifFuzz will be given, as these are the three main fuzzers we have used and based our work upon. Section 3.2 will describe side-channel analysis from a high-level standpoint and emphasize the relevance to our experiments.

3.1 Fuzzing

The term fuzzing refers to the technique of injecting invalid or malformed inputs into a system using an automated software. A fuzzing tool will monitor the system's behavior to these inputs and record any crashes, information leakages, software defects and vulnerabilities that will occur.

Before providing more details, we will give a trivial example about how the fuzzing process works. Assume we have a program that uses an integer variable to store the result of a user's choice between 3 di erent questions. When the user picks a value for the question, the program expects the value to be `0`, `1` or `2`. However, what if we transmit `-3`, or `255`? This is possible, as an integer can take any value between `-2,147,483,647` and `2,147,483,647` in most computers and programming languages. If the program does not address this issue, it may cause crashes or lead to security issues: (un)exploitable bu er over ows, Denial of Service attacks etc.

3.1.1 Concepts

As explained in the de nition above, the goal of fuzzing, or fuzz testing, is to introduce inputs that are intentionally malformed into a system. In this way, failures or defects that exist within the targeted system will be identi ed. Every fuzzer, or fuzzing tool, has three key components: a poet, a courier and an oracle. These are de ned below.

The poet is the process starter. It creates test cases that are going to be used on the target system. These cases are based on the type of fuzzing that is being performed, which can be de ned into *generational*, *mutational*, *evolutionary*, and *di erential* fuzzing. These will be explained in detail in Section 3.1.2 [1].

The courier, as the name suggests, is supposed to deliver the cases to the target system. The delivery process varies per type of fuzzing that is being performed, however details are not relevant for our work [1].

The oracle has as main goal the monitoring of the test cases as well deciding whether a test has passed or failed. Checking for a failure within the system is important, as otherwise the developers of the system would not be able to identify and x it [1].

3.1.2 Techniques

Depending on level of analysis and dependency on the source code, fuzzers can be classi ed into white-box, gray-box and black-box fuzzers. White-box fuzzers have access to the complete source code of the target system. Therefore, they are capable of performing an analysis on the code and derive information about the e ects of the test cases on the program state. Black-box fuzzers can only query the target system and record the response. Thus, they do not have access to the source code and cannot perform an analysis on it. Gray-box fuzzers also cannot access the source code, however they can perform an analysis during the execution of the program [10].

As mentioned in Section 3.1.1, fuzz testing can also be divided into generational, mutational, evolutionary, and di erential fuzzing. We will now explain these notions in more detail.

[1] While not present in any scienti c paper that we have cited throughout our work, we consider this division to be easy-to-understand for anyone who is not experienced with fuzzing tools. This information is taken from: https://www.synopsys.com/glossary/what-is-fuzz-testing.html

Generational fuzzing uses inputs generated from scratch. Therefore, these fuzzers do not need a set of input files. They will use the known, predefined format, and will generate new inputs based on this format. Mutational fuzzing uses an existing set of (valid) inputs and modifies these throughout the process.

Evolutionary fuzzing is similar to mutational fuzzing. The difference is that it uses various genetic algorithms to create less but better inputs for the target system. The evaluation of these inputs is being done by a *fitness* function. Only the inputs that pass this fitness evaluation will be used for further generation [17]. The fuzzer we have used for our initial attempts at creating a fuzzing tool that uses timing side-channel information, AFL, is an example of a evolutionary fuzzer.

Differential fuzzing is a newer type of fuzz testing, which is based on a technique called differential testing. This type of testing compares a programme's behavior to a certain, set baseline and draws conclusions based on this comparison. Similarly, with a differential fuzzer, multiple programmes are tested with the same input and the differences are being analyzed. DiffFuzz is an example of a differential fuzzer.

3.1.3 AFL

Even though fuzzing is a well known technique of testing software programs, most fuzzers are relatively shallow. They make random mutations to the input files and thus it makes it very unlikely to find certain paths that might contain vulnerabilities. AFL solves this problem by using an instrumentation-guided genetic algorithm, which is able to detect well-hidden branches or changes to the program control flow [7]. In the following sections we will explain the most important technical details of AFL, as they are described in its technical paper [20].

High-Level Description

While the internals of the fuzzer are more complex, the steps that the overall algorithm follows can be summarized as follows [7]:

1. The initial inputs, which are provided by the user, are loaded into the queue.
2. The next input is taken from the (the front of the) queue and given to the fuzzer.
3. The fuzzer attempts to reduce the input's (or test case's) size using trimming as long as the behavior of the program does not change. Once this happens, this means that the smallest size has been reached.

4. The input is repeatedly mutated by the fuzzer, using a variety of traditional fuzzing strategies. Note that here the mutations applied on the input are independent of each other, as they are not chosen from a sequence of mutations.

5. If one of the above mutations results in new edge coverage (which is being recorded by the inserted instrumentation), the mutated output becomes a new entry in the queue, which is added to the front.

6. Go back to step 2.

Fuzzer Modules

The AFL fuzzer combines many different working parts, however based on the high-level description we presented in Section 3.1.3 we can create an abstraction and divide AFL into four main modules, which can also be seen in Figure 3.1. Note that many other fuzzers have similar functionality, and thus this division is generalizable.

1. The queue module. It ensures that the many (mutated) inputs are properly ordered, keeps track of the favored inputs, removes the unnecessary or superfluous inputs and reduces the size of the inputs. Part of the poet and the courier, as described in Section 3.1.1.

2. The mutations module. Its main role is to generate mutations for the input that is currently used by the program under test. The mutations include various fuzzing strategies, however the details are not important. Part of the poet, as described in Section 3.1.1.

3. The shared memory or bitmap module. Its purpose is to monitor and store the branches that are being reached during fuzz testing by the different inputs. The process is explained in slightly more details later in this section. Part of the oracle, as described in Section 3.1.1.

4. The instrumentation module. It guarantees the generation of the instrumentation that will be added to the program that is being fuzzed. The main role of the instrumentation is to record the branch jumps and report these back to the fuzzer. Part of the poet, as described in Section 3.1.1.

Additionally, we consider modules 1 to 3 to be part of the main code of AFL, and that module 4 can be further divided into two submodules, which again can also be seen in Figure 3.1.

a. The code that generates the instrumentation applied to the System Under Test (SUT). This is part of the AFL implementation itself.

b. The actual instrumentation applied on the SUT. This is additional code added to the SUT in order to monitor branch jumps during program execution.

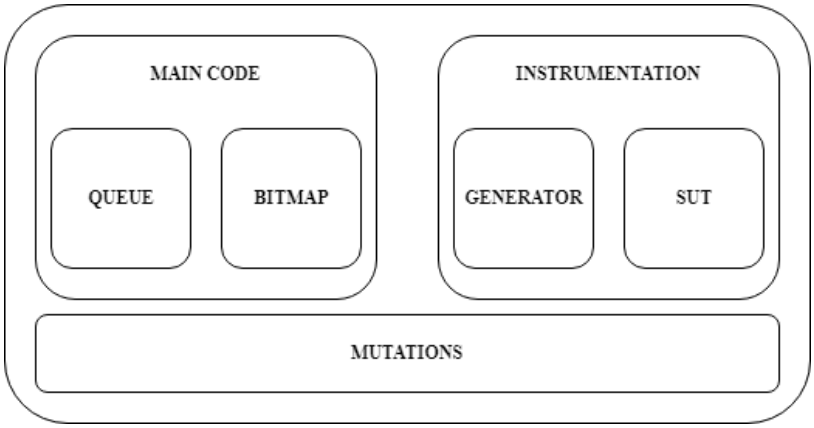

Figure 3.1: AFL Module Structure

Edge Coverage Measurement

In order to measure how many times a branch (or edge) is reached, AFL makes use of what is called instrumentation. To this extent, the following piece of code is inserted at branch points [20]:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Code 3.1: Instrumentation code inserted by AFL at branching points [20]

The `cur_location` value is set randomly at compile time as this allows for uniformity to be kept for the XOR operation. `shared_mem` is a 64Kb array, which is seen as a byte map by AFL. This means that every byte being set or incremented in the byte map represents a hit for the tuple <`src_branch`, `dst_branch`>. Such an implementation is used as it offers a more fine-grained analysis of the execution path of a program, e.g., it can

distinguish between a path that has the same starting and ending branch, but di erent intermediary jumps [20]. Take as an example the following two execution paths:

1. `A->B->C->D->E->F`
   This contains the branch tuples `AB, BC, CD, DE and EF`.

2. `A->B->D->C->E->F`
   This contains the branch tuples `AB, BD, DC, DE and EF`.

These two execution paths are seen as being di erent by AFL, even though both start at branch `A` and end at branch `F`. This allows for the discovery of vulnerabilities that are much more hidden in the underlying code, and will be harder to detect just with simple block coverage. To illustrate the memory layout of `shared_mem`, take a look at Figure 3.2.

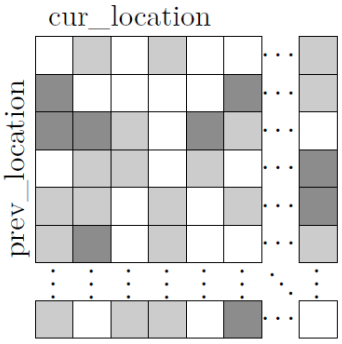

Figure 3.2: AFL Shared Map Implementation

Detection of New Behaviors

As explained in Section 3.1.3, the fuzzer maintains a shared, or global, map of previously discovered execution paths, which can be e ciently updated in a few instructions. This map allows for a better mutational process of inputs. If a certain input le produces new tuples in the execution path, it will be saved and given to additional processing. If no new tuples are discovered, the input le will simply be discarded (see step 5 in Section 3.1.3). As we have previously explained, this approach conducts a more ned-grained analysis of the program execution, while avoiding any intensive computations [20].

To illustrate this approach, we consider the following two execution paths:

1. `A->B->C->D->E->F`
   This contains the branch tuples `AB, BC, CD, DE and EF`.

2. `A->B->C->D->A->F`
   This contains the branch tuples `AB, BC, CD, DA and AF`.

Path #2 is seen as being substantially di erent than path #1 because of the tuples `DA` and `AF`. Therefore, once #2 has been registered within the shared memory, the following path will not be seen as unique:

3. `A->B->C->D->A->B->C->D->A->B->C->D->E->F`
   This contains the branch tuples `AB, BC, CD, DA, DE and EF`.

This is because the tuples that are present within path #3 have been seen before by the fuzzer and stored in the shared memory [20].

3.1.4 LibAFL

The idea behind LibAFL lies within the fact that most fuzzers, while e cient and e ective, are not easily extensible or modi able. This means that, in most cases, a developer or researcher either has to fork one of these existing fuzzers, or create one from scratch. This results in a lot of unecessary work and incompatibility issues between fuzzers. LibAFL aims to solve this issue by o ering a \collection of reusable pieces for individual fuzzers" [3]. This makes the process of creating a fuzzer much more modular and straightforward.

As LibAFL is a collection of di erent features from di erent fuzzers, there are many combinations that can be created. Therefore, o ering a high-level description is not necessary in this section. However, LibAFL consists of di erent components, or modules, which we will describe in short based on their book [3]. More details about the various implementation options for these modules and their functionality can be found at the LibAFL docs [4] or the LibAFL research paper [5].

Core Components

The LibAFL architecture can be divided into 8 core concepts or entities, which can also be used as an abstraction for other fuzzer designs, as mentioned in the LibAFL book [3]. This architecture has similarities with our own abstraction of AFL's fuzzing process, as described in Section 3.1.3.

- The Observer. As the names suggests, the role of the Observer is to provide the fuzzer with information gathered during the execution of the program being fuzzed. The observations can consist of a coverage map, execution time or maximum stack depth, amongst others.
- The Executor. This entity's main role is to de ne how to execute the target program. Moreover, the Executor is also responsible with informing the target about the volatile operations related to a single run, such as the input being used or writing to a certain memory location. An Executor can also hold a set of Observers which gather information about each program run.
- The Feedback. The role of the Feedback entity is to classify whether or not an execution of the program being fuzzed is interesting or not. If the execution was interesting, then the respective input is added to a corpus. What is meant by \interesting" is abstract, but, for example, an input that results in a higher execution time than the current average can be considered interesting. A Feedback entity is strongly related to the Observer entity.
- The Input. In an abstract view, the Input entity is the internal representation of the program input. In the most common case, the input of the program is a byte array, however, more complex representations can also be used.
- The Corpus. Simply put, the Corpus is the entity where all testcases are stored. A testcase is de ned as \an Input and a set of related metadata like execution time for instance" [3]. A testcase is added to the Corpus if it is considered interesting by the Feedback entity.
- The Mutator. This entity's role is to generate new instances of Input derived by mutating the current Inputs.
- The Generator. A Generator is used to generate an instance of Input from scratch, usually by utilizing a random number generator.
- The Stage. This is an entity that applies a Mutator on an Input from the Corpus. How many times this is done, can be de ned using various parameters.

3.1.5 DifFuzz

As explained in Section 2, DifFuzz is a newer fuzzer which allows for the discovery of side-channel vulnerabilities using differential fuzzing. Therefore, the main idea is that DifFuzz divides the program under test, say f, into two copies, using the same values for the public input, say pub, but different values for the secret input, say sec_1 and sec_2 respectively. Therefore, the input for f is the tuple (pub, sec_1, sec_2), and the two copies of f are $f(pub;\ sec_1)$ and $f(pub;\ sec_2)$. Then DifFuzz computes the difference over the side-channel measurements (either time or space) between the two executions, which will be treated as a score. If the resulting score is large, then a side-channel vulnerability is present.

High-Level Description

The fuzzing approach used by DifFuzz can be summarized as follows [15]:

1. The initial inputs, which are provided by the user, are loaded into the queue.
2. The user then needs to provide a driver, which parses the input file(s) into (pub, sec_1, sec_2). The driver also executes $f(pub;\ sec_1)$ and $f(pub;\ sec_2)$. Providing the driver is an extra step from AFL's workflow, as explained in Section 3.1.3.
3. The next input is taken from the (the front of the) queue and given to the fuzzer.
4. The input is repeatedly mutated by the fuzzer, using a variety of traditional fuzzing strategies. Note that here the mutations applied on the input are independent of each other, as they are not chosen from a sequence of mutations.
5. If one of the above mutations results in new edge coverage or higher score (which are being recorded by the inserted instrumentation), the mutated output will become a new entry in the queue, which is added to the front.
6. Go back to step 3.

The overview of this approach is shown in Figure 3.3 There are two important things to note here. Firstly, DifFuzz makes use of custom instrumentation to record side-channel information - e.g., execution time or memory consumption. This is important as in Section 4 we will explain how this is different from our approach. Secondly, the driver is target specific, and thus a driver has to be manually created for each of the programs that we want to fuzz.

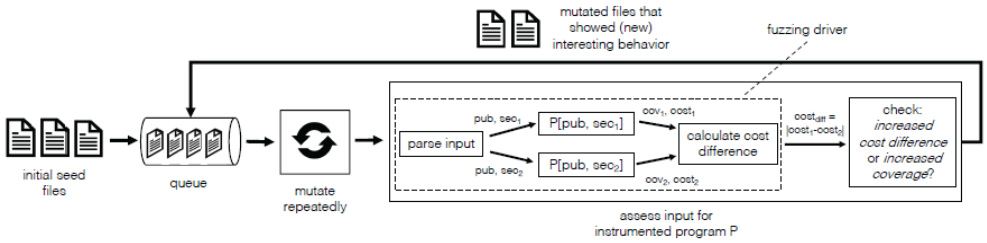

Figure 3.3: High-Level Description of DifFuzz fuzzing process

3.2 Side-Channel Analysis

Side-Channel Analysis aims to investigate whether or not cryptographic algorithms or programs working with secret data, which are running on a hardware device, leak any sensitive information via side-channels. A side-channel can refer to timing information, power consumption, or electromagnetic radiation, amongst others. Similarly, side-channel attacks aim to exploit such vulnerabilities by finding correlations between the information leaked through a side-channel and the secret data.

An important property used in side-channel analysis is **non-interference**, which means that a program is secure if no secret information can be inferred from observations, such as power consumption, made from the target hardware device. One of the techniques used to check this property is called **self-composition**, which is what both DifFuzz [15] and QFuzz [16] are using. The details of how this technique works are not relevant for our thesis.

As described in Chapter 1, our end goal was to use our custom fuzzer on smart cards. To this extent, the most relevant side-channels are **timing leaks** and **power consumption**. Timing leaks side-channels are based on the execution time of a program, while power consumption side-channels make use of the power usage information during the execution of a program. It is important to note that power consumption is an implementation-specific characteristic, as it depends on the data and operations that are being executed. Therefore, being able to use such a side-channel could potentially improve the efficiency of a fuzzer.

Chapter 4

Design Concept for using Timing Side-Channel Information during Fuzz Testing

In Chapter 1, we stated our objective for this thesis, which is to create a fuzzer that uses timing side-channel information to guide the fuzzing process towards finding the secret data being used by the program being fuzzed and eventually toward improving code coverage. As we also mentioned, we will call this fuzzer SCFuzz. SCFuzz is the first attempt at creating a fuzzer for these purposes.

In the next sections we will present various decisions and notions that build up our design process for using timing side-channel information during fuzz testing. In Section 4.1 we present the main design decisions we have made while working on our implementation. Section 4.2 will explain the purpose of a score when implementing timing side-channel measurements during fuzz testing as well as the different ways in which the score can be computed in our implementation of SCFuzz. In Section 4.3 we offer an overview of the steps taken by SCFuzz during the fuzzing process. Section 4.4 contains the password checking program which is vulnerable to a timing side-channel, as well as a brief explanation of certain decisions we have made about this program. Here we will also present a short analysis of the results obtained after fuzzing the program using AFL, which will give us a baseline for the results obtained with SCFuzz. Section 4.5 summarizes the main decisions from the design process as well as offers an overview of the different SCFuzz versions, which we will present in more detail in the next chapters.

4.1 Design Decisions

In order to implement side-channel information into the fuzzing process, we first need to decide what side-channels we should use. As explained in Section 3.2, there are multiple options. We have chosen **execution time** as a side-channel for our initial SCFuzz attempts, as this is the easiest side-channel to implement - we can simply record the the time when the execution starts and the time when the execution ends, and record the difference. We will additionally briefly describe how **power consumption** can be used as a possible side-channel for the fuzzing process in future work.

In Section 3.1.3, we presented the four modules in which the AFL fuzzer can be divided into. However, we also mentioned that this division can be similarly applied to other fuzzers. Using this division, there are two possibilities in which we can modify an already existing fuzzer - or create our own - to use timing side-channel information for guiding the fuzzing process:

1. Modify the instrumentation which is applied on the System Under Test (SUT) to record the program execution time.
2. Modify the fuzzer's main code such that it measures the time between when the input is sent to the SUT and when the SUT outputs results.

DifFuzz uses approach 1, as they measure execution time by counting (byte) instructions, since it is built on top of the Kelinci-WCA. Kelinci-WCA is a fuzzer interface which allows for AFL to be used on Java programs, and thus both DifFuzz and Kelinci-WCA use custom instrumentation for edge-coverage. Our end goal was to test SCFuzz on smart cards, which do not allow for instrumentation to be added to the code running on them. Thus, this approach would be unfeasible in our case. We decided to go with the **approach 2** (which is also confirmed in the DifFuzz paper as being a valid option [15]).

To guide the fuzzing process using timing side-channel information, we need to ensure that we properly store and interpret the side-channel measurements, e.g., the execution time of a program, in our case. This can be achieved by making use of a **score** value which will guide the fuzzer into, for example, finding the secret data. We will further illustrate this in Section 4.2.

To make it easier for us to analyze the SCFuzz implementation, we want to have a relatively simple program which is vulnerable to a timing side-channel. We decided to use a password checker which compares the user input with the correct password character by character, and thus has a timing leak. We will further explain this program in Section 4.4.

Since we are incrementally going to create our fuzzer, we make the following assumptions with regard to the attacker model for the initial SCFuzz versions, as presented in Chapter 5 and 6:

1. We know the program we are fuzzing has a timing vulnerability
2. We know what is the functionality of the program we are fuzzing
3. We can use program specific information in the fuzzing process

Thus, we will be using a white-box approach, as described in Section 3.1.2. These assumptions ease up the process of creating an initial implementation of SCFuzz, as there is more information we can use during the fuzzing process. For example, we can implement a pseudo-timing side-channel using return codes or create a dictionary such that the mutations applied on the test cases only result in strings containing ASCII characters, when fuzzing the vulnerable password checker program presented in Section 4.4. Our end goal was for our final version of SCFuzz to only use the actual execution time of the program, and thus use more of a gray-box approach, and thus we will shortly discuss such an approach in future work.

There are also two possible options for creating the actual implementation of SCFuzz. We can use an already existing fuzzer with its complete functionality, e.g., edge-coverage instrumentation for AFL, and simply add the necessary changes to include side-channel information, or we can make our own fuzzer. Throughout our thesis we have explored both possibilities, which will be described in more detail in Chapter 5 and Chapter 6, respectively.

4.2 Timing Side-Channel Score

As mentioned in Section 4.1, in order to guide the fuzzing process using timing side-channel information, we need a score value. Since we are using the execution time of a program as a side-channel for our implementations of SCFuzz, we need to find a way to attribute a score with regard to the program execution time to each entry in the input queue. This score will then be used by the fuzzer to keep interesting inputs and mutate them further.

There are two reasonable ways to compute the score of the individual inputs with regard to the execution time. Before we can formalize these methods, we need some variables.

- i: The input used by the current execution of the program being fuzzed.
- $desc_i$: A descendant of input i, obtained by mutating input i.
- D: The set of all descendants $desc_i$ for a certain input i.
- $score_i$: The score attributed to input i after it has been processed.
- $start\_time\_i$: The exact time at which the execution of the program being fuzzed with input i has started.
- end_time_i: The exact time at which the execution of the program being fuzzed with input i has nished.
- $diff(A; \forall s \in S)$: The function $diff$ computes the number of di erent results after applying computation A on all elements s in set S.
- $t(x; \ )$: The function t evaluates to `true` if value x has not changed after seconds.

The two options for computing the score are formalized below.

1. High-score Approach
   $score_i = end\_time_i - start\_time_i$
   `if` $score\_i \geq high\_score$ `then` $high\_score = score\_i$
   SCFuzz will thus only keep (and mutate) the input i if it results in a score higher or equal to the current $high\_score$.
2. Unique Descendants Approach
   $score_i = diff(end\_time_{desc_i} - start\_time_{desc_i}; \forall desc_i \in D)$
   `if` $t(score_i; \ )$ `continue with next` i
   In other words, $score_i$ is computed by counting the number of descendants that result in a di erent execution time. When no new execution times are discovered from the current input after seconds, we proceed with the next input.

Because of the assumptions described in section 4.1, and because it is much easier to implement, we have decided to proceed with the High-score Approach for our implementations of SCFuzz, applied on the vulnerable password-checker program. However, as mentioned in 4.1 as well, that means our implementations are going to use a white-box approach, as we know the program will take longer to execute when the fuzzer nds more correct characters for the password. We will discuss how could the second the Unique Descendants Approach be a better option and thus worth exploring in future work.

4.3 High-Level Description

Similarly to the fuzzing process of AFL and DifFuzz, which were described in Section 3.1.3 and Section 3.1.5 respectively, the steps for the overall algorithm of SCFuzz are as follows:

1. The initial inputs, which are provided by the user, are loaded into the queue.
2. The next input is taken from (the front of the) queue and given to the fuzzer.
3. The input is repeatedly mutated by the fuzzer, using a variety of traditional fuzzing strategies. Note that here the mutations applied on the input are independent of each other, as they are not chosen from a sequence of mutations.

4a. If the current input results in a higher score than the current highest value for the score, we will keep this input in the queue for further mutations and go back to step 2 (Option 1 - High-score Approach).

4b. If $t(score_i;\ )$ evaluates to `true` for the current input, go back to step 2 (Option 2 - Unique Descendants Approach).

4.4 Password Checker with (Pseudo-)Timing Side-Channel

As explained in Section 4.1, we will make use of a password checking program which is vulnerable to a timing side-channel, in order to test our initial implementations of SCFuzz. This program is written in C, and can be seen in Code 4.1.

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

int main(int argc, char *argv[]) {
    char pub[7];
    char sec[7]="mysecurepassword123";
    int pub_len, sec_len = 7;
    char ok;

    if(argc < 2) {
        printf("No argument passed through command line.\n");
        return 0;
    }
    else {
        printf("First argument is: %s\n", argv[1]);
    }
```

```c

    FILE* fptr = fopen(argv[1], "r");

    if(fptr != NULL) {
        pub_len = fread(&pub, 1, 7, fptr);

        if(fread(&ok, 1, 1, fptr) != 0) {
            printf("Password too long.\n");
            return 2;
        }

        pub[pub_len - 1] =  \0 ;
        printf("%s \n", pub);

        fclose(fptr);
    } else {
        printf("Not able to open the file.\n");
        return 1;
    }

    if (pub_len != sec_len) {
        printf("Lengths do not match.\n");
        return 3;
    }

    for (int i = 0; i < pub_len; i++) {
        if (pub[i] != sec[i]) {
            printf("Wrong character.\n");
            return i + 4;
        }
        sleep(1);
    }

    printf("Correct password.\n");
    raise(SIGSEGV);
    return 400;
}
```

Code 4.1: Password Checker with Timing Leak - C

There are some important attention points about this vulnerable password checking program:

- We have di erent return codes for all the possible cases at which the program can end.
- The return codes are in incremental order such that the fuzzer can be guided towards nding the correct password (see assumptions in 4.1). We can think of these return codes as a pseudo-timing side-channel.
- We raise an error when the correct password is entered to more easily see when this case is reached by SCFuzz within the status screen.
- We ensure that there are no other vulnerabilities, such that no additional crashes can occur when fuzzing this program. In this way, we can con rm that our crash is indeed caused by the correct password being found.

SYSTEM SETUP	
Operating System	`Kali 2022.4 (Virtual Machine)`
CPU	`AMD Ryzen 7 5800H, 3.2 GHz, 8 Core`
RAM	`16 GB`
GCC Version	`v13.2.0`
Rust Version	`v1.74`
AFL Version	`v2.51b`

Table 4.1: The system setup that was used throughout our thesis

- To the extent possible, with the exception of the loop iterating through the characters of the user password (lines 43 - 48 in Code 4.1), we attempted to make the program constant time, such that minor di erences in execution time will not cause SCFuzz to lose the path to the correct password. However, this is not something we can fully prevent.

4.4.1 Baseline using AFL

In order to have a baseline for the results we obtain using SCFuzz, we fuzz the vulnerable password checking program using an unmodi ed version of AFL. The system setup is given in Table 4.1 and the results are shown in Table 4.2. As can be seen, AFL is capable of nding the correct password, without any modi cations for using side-channel information, assuming the program will crash once this password is given. However, it is important to note that this is possible, in part, due to the way we have implemented our password-checking program, e.g., each possible program state has its own unique branch, and thus AFL's edge-coverage algorithm and instrumentation is able to identify these di erent branches, as explained in Section 3.1.3. Moreover, in the case of the loop on lines 43 - 48 from Code 4.1, AFL is not able to make any inferences about how many characters it guessed correctly, as nding a correct character will not result in a di erent branch being discovered. This is also proven by the output of the AFL terminal seen in Figure 4.1. There we can see 8 di erent execution paths are found (*total paths* entry on the status screen), which is the same as the amount of di erent program branches in Code 4.1. Since the mutations applied by AFL on the inputs are random, the fuzzing process at that point in the program is thus also random, and not guided by how many characters have been guessed correctly. This can again also be seen in the results shown in Table 4.2, as the amount of time it takes for AFL to nd the correct password varies greatly between runs and no clear conclusions can be made, e.g., longer passwords do not necessarily take longer to be found. The average time seems to be slightly better corellated to the password length, but even here, `myse` has a smaller value for the average time than `mys`.

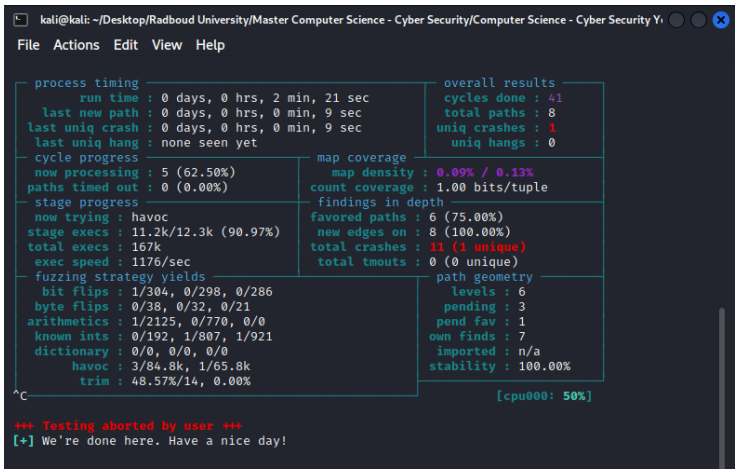

Figure 4.1: AFL Status Screen running on the Password Checker with Timing Leak - C Program, for password *mys*

Another important observation is that AFL was not able to find the correct password for the last fuzzing attempt presented in Table 4.2. When the correct password is *mysecu1*, AFL was not able to find this password after 30 minutes of fuzzing for all five fuzzing runs. We have decided to create this fuzzing attempt as a way to mitigate a possible bias that can appear when the characters of the correct password are only literals, a bias that can also explain the absence of correlation between password length and time it takes AFL to find the password. Explaining this behavior is out of scope for this thesis.

Based on the preliminary findings explained above, we expect that SCFuzz will be able to find the correct password regardless of the format of the password string, and moreover that there will be a stronger correlation between password length and the amount of time it takes the fuzzer to find the correct password. This in turn will also prove that the changes we make for implementing side-channel information in the fuzzing process actively guide the fuzzer towards finding the secret data. Additionally, we also expect that, based on the number of different execution times found during the fuzzing of a program with SCFuzz and based on their respective inputs, we are able to replace the instrumentation module, as described in Section 3.1.3, for code coverage, using the Unique Descendants Approach.

PASSWORD	ATTEMPT #1 TIME	ATTEMPT #2 TIME	ATTEMPT #3 TIME
my	1 min 2 sec	22 sec	13 sec
mys	2 min	2 min 12 sec	5 min 22 sec
myse	9 min 10 sec	1 min	22 sec
mysec	3 min 4 sec	5 min 1 sec	10 min 50 sec
mysecu	3 min 51 sec	11 min 20 sec	30 min (!)
mysecu1	30 min (!)	30 min (!)	30 min (!)

ATTEMPT #4 TIME	ATTEMPT #5 TIME	AVERAGE TIME
14 sec	7 min 9 sec	1 min 48 sec
1 min 50 sec	16 sec	2 min 20 sec
31 sec	20 sec	2 min 16 sec
19 sec	5 min 3 sec	4 min 51 sec
7 min 1 sec	15 min 56 sec	9 min 32 sec
30 min (!)	30 min (!)	-

Table 4.2: Total time to find the correct password when using plain AFL on the vulnerable password checking program. The first column shows the password we are trying to find, and the other six columns show how long it took AFL to find that password as well as the average time for the five attempts, respectively. The symbol (!) means that the fuzzing attempt was unsuccessful in finding the correct password, and the average time does not include the unsuccessful attempts

4.5 Summary

This chapter contains various decisions that build up our design process for using timing side-channel information during fuzz testing. We will summarize these decisions below:

1. We use execution time as a side-channel for the SCFuzz implementations seen in Chapters 5 and 6.
2. We measure execution time using approach 2, as seen in Section 4.1.
3. To use side-channel information within fuzz testing, we need a score and a way to process this score. We will be using the High-score Approach for the SCFuzz implementations described in Chapters 5 and 6, as explained in Section 4.2, while the Unique Descendants Approach will be left for future work.
4. The SCFuzz implementations described in Chapters 5 and 6 are applied on the password checker program seen in Section 4.4, program which has a timing leakage.
5. The results obtained using the SCFuzz implementations described in

Chapters 5 and 6 are compared to the baseline result obtained using AFL, as seen in Section 4.4.1.

6. The SCFuzz versions described in Chapters 5 and 6 implement a white-box approach, using the assumptions explained in Section 4.1.

7. We can implement SCFuzz by either modifying an already existing fuzzer, or by making our own. Thus, we have the following SCFuzz versions:

    - $\text{SCFuzz}_{\text{AFL}}$: This version uses AFL's complete underlying functionality, e.g., the instrumentation used for edge-coverage, in addition to the timing side-channel information. Described in Chapter 5.
    - $\text{SCFuzz}_{\text{LibAFL}}$: This version uses only basic fuzzer modules, and we add our own functionality using the timing side-channel information. Described in Chapter 6.

Chapter 5

SCFuzz$_{\mathbf{AFL}}$

As mentioned in Chapter 4, our first implementation of SCFuzz will use AFL as the underlying fuzzer, together with our changes, which implement the design decisions as described in Section 4.1, to include timing side-channel information. We decided to take this approach first, as the modifications that are required are similar (and based on) the DifFuzz implementation, which also uses AFL. Moreover, since the instrumentation can be disabled in AFL, it would not result in issues when working with smart cards.

In Section 5.1, we describe the functions in AFL's code that we are interested in and the required modifications to these functions such that execution time can be used as a timing side-channel in the fuzzing process. Section 5.2 explains why we initially opted for AFL++, and what issues we have encountered that caused us to not continue with this version. In Section 5.3 will present various attempts at creating a version of SCFuzz using AFL and the respective results. Moreover, we will also offer an explanation of why we believe we were unsuccessful in creating this version. Section 5.4 summarizes the main takeaways from the implementation process.

5.1 AFL Functions and Modifications

The main issue with regard to integrating timing side-channel information in AFL is how to properly store the score value. We discovered that there are two ways in which the timing information can be added to AFL, based on how AFL stores information between all the executions of the program being fuzzed:

1. As explained in Section 3.1.3, AFL makes use of a shared memory array, which we call `shared_mem`, to store the edge coverage information. We could use the same array to additionally store information about timing on top of branch jumps, using a transformation function.

2. AFL makes use of a queue to store the (mutated) inputs. Each of these inputs is stored in the queue using a `queue_entry` structure, which stores various information about the results obtained after executing the program under test with this input, e.g., whether or not an input triggers new coverage. We can thus create an additional field in this `queue_entry` structure, which stores a score relating to the execution time for each entry. Thus, a queue entry will become a tuple $(i;\ execution\_info;\ score_i)$.

We concluded that the first approach would not be ideal, as the shared memory code is strongly connected to the instrumentation code, and thus modifying it would not be straightforward. Moreover, the shared memory would not be enabled if we do not use the instrumentation - which is the goal, as we aim for the execution time information to replace the instrumentation for edge coverage. Therefore, we have decided to proceed with the second approach. For this, we need to describe the AFL functions that require modifications, and the respective changes that we have to make to them, based on the design decisions presented in Section 4.1:

- We need to initialize some global variables and `queue_entry` variables, such that we can store the (highest) score between program runs as well as the score for each fuzzing input.
- The function `update_bitmap_score`, which checks whether or not a new path is more favorable - in terms of the bits set in the shared map - than any of the other existing ones. We need to modify this function such that a more favorable path means that it results in a higher timing side-channel score.
- The function `cull_queue`, which ensures that favored queue entries are selected to get more air time during fuzzing. Similarly, we need to modify this function such that the selection is achieved using the timing side-channel score.
- The function `calibrate_case`, which aims to detect problematic inputs early on. Because of this, the function also executes the program being fuzzed with the inputs, and thus we need to ensure that the execution time is recorded.
- The function `save_if_interesting`, which saves a certain input if it results in an interesting outcome after execution. We need to modify it such that it additionally takes into account whether the input results in a new high score or not.
- The function `common_fuzz_stuff`, which executes the program being fuzzed with a certain input. We need to ensure that the execution time is recorded here.

- The function `calculate_score`, a helper function for the mutations module. Not important how it exactly influences the fuzzing process, but we need to ensure it also includes the timing side-channel score.
- Finally, we need to create a couple new functions for processing the scores and saving the high score.

The complete modifications made to AFL are shown in Appendix A.

5.2 SCFuzz using AFL++

First, we looked into creating an SCFuzz implementation using AFL++, using the changes described in Section 5.1. Because AFL is no longer maintained, and because AFL++ offers the same functionality as well as improvements with regard to efficiency and input generation, we decided it would be a good option for our work. Since we based our approach on DifFuzz, which is built on top of the Kelinci-WCA fuzzer interface which allows for AFL to be used on Java programs, modifying AFL++ would not require much extra effort. However, after spending more than three weeks on this attempt, we unfortunately had to scratch our work on AFL++ as we failed to implement all the necessary changes. The functions described in Section 5.1 are more complex than in the older AFL and the code, while technically more clean, consists of too many files with code that shares functionality and data. Because of this, we decided to proceed with the older version of AFL.

5.3 SCFuzz with AFL

Secondly, we looked into creating an SCFuzz implementation using AFL, using the changes described in Section 5.1. In order to modify AFL using those changes, we are interested in the file `afl-fuzz.c`, as this file contains the main part of the binary fuzzing tool. The exact changes can be found in Appendix A.

5.3.1 First Attempt: AFL Instrumentation & Execution Time as a Side-Channel using the High-score Approach

Initially, we have implemented the changes as described in Section 5.1, without using the output of the program under test. This is because we wanted to see if our current approach of computing and using the execution time as a score would be enough to find the correct password. Moreover, we also wanted to keep the instrumentation for the initial attempt, as it would make it easier to compare it to the baseline results presented in Section 4.4.1 and

thus see if our changes have any positive e ect. However, after implementing our changes, we ran the fuzzer for about 15 minutes and arrived at the following conclusions:

- Since we are using nanosecond precision, there is a lot of noise and thus there are too many new high scores found.
- While the goal of the fuzzer is to maximize the execution time, as this will technically lead to the correct password being found, without additional information this approach does not seem feasible.

Based on these ndings, the next step was to use the return codes of the program as well as add a variance threshold such that the number of di erent execution times is reduced.

5.3.2 Second Attempt: AFL Instrumentation & Execution Time as a Side-Channel & Response Codes as a Pseudo-Timing Side-Channel using a High-Score Approach

Our next option was to attempt to guide the fuzzer using the return values of the password checker in addition to the execution time values. As can be seen from Code 4.1, depending on the conditional branch being reached, there are di erent return values being returned. Since these return values are also being implemented in incremental order, with the highest return value, e.g., return value 400, representing that the correct password has been given as input, our idea was to guide the fuzzer by only keeping inputs that result in a higher return value in combination with our current implementation of the score.

Additionally, we also added a threshold for the execution time, such that only those new execution times that exceed this threshold are going to be added to the queue by the fuzzer. In other words, the *score* value for a certain input will be kept if and only if $score \geq high\_score + threshold$. We have done this in order to mitigate possible minor di erences in execution time when certain inputs are used, but are not directly related to the amount of characters that have been guessed correctly, e.g., small di erences when the password is being read from the le. This is in addition to what we have already done in the vulnerable C program itself to tackle this issues, as explained in Section 4.4.

The combination of using execution time and response code results has resulted in our fuzzer to successfully nd passwords shorter than 5 characters. The crash represents the fact the the correct password has been found, which is a result of the `raise(SIGSEV)` call at the end of the program, as shown in Code 4.1. However, for longer passwords, our fuzzer was again not able to nd the correct password after 15 minutes.

In order to discover where does the problem exactly arise from, we have also tried using the following changes:

- In the previous attempts, we have been measuring the execution time of the program under test around the `run_target` call in the fuzzer. However, there are still additional operations being done outside of executing the program under test in `run_target`, which can add noise. Therefore, we have tried to measure the execution time of the program being fuzzed as close as possible to the `execve()` call in the fuzzer.
- In the previous attempts, each time the password contained a wrong character, we were returning the value 3 regardless of the index of the wrong character. Therefore, in a different attempt, we have returned the value `i + 4` on line 46 of Code 4.1, such that we can guide our fuzzer to find the correct password character by character.
- Finally, we have also attempted to add a `sleep` call after each correct character in the password, with a high value, e.g., 1 second. In this way, we not only try to mitigate the noise that we have mentioned, but also again try and guide the fuzzer to find the correct password character by character by ensuring bigger timing differences if a character check turns out successful.

Unfortunately, even with the extra changes we have presented above, we did not manage to get our fuzzer to find passwords longer than or consisting of 5 characters.

5.4 Conclusions

The main conclusion that we can derive from our attempts at creating our initial implementation of SCFuzz using AFL++ and AFL is that we were not successful in modifying either AFL++ or AFL in order to guide the fuzzing process using the execution time of the program under test toward edge-coverage or toward find the correct password for a vulnerable C program. As we have described in this chapter, we have made various attempts at creating this version of SCFuzz, however we did not manage to have our fuzzer find passwords longer than or consisting of 5 characters, regardless of how much (pseduo-)side-channel information or fine-tuning we have used to guide the fuzzer. Our guess is that either due to the random mutations which AFL applies to the inputs or because AFL is blind to what the execution time values and the return code values are actually supposed to represent, it is not possible to add side-channel information to AFL without going much deeper into its functionality. However, as mentioned, this is just a hypothesis that we have not explored further, as this was not the main goal of our thesis.

Chapter 6

SCFuzz$_{\text{LibAFL}}$

Since our attempts to modify AFL and AFL++ have failed, as discussed in Chapter 5, our second implementation of SCFuzz uses LibAFL, a library containing various fuzzer modules. LibAFL makes the process of creating a fuzzer from scratch more modular and less tedious, as we described in Section 3.1.4. Thus, we can simply pick the components and features we are interested in, and then incrementally add them together. The available components are again described in short in Section 3.1.4. Since LibAFL already has the basic features that our fuzzer implementation requires, including modules that allow us to include execution time as a side channel, we hope that the implementation process will be rather straightforward.

In Section 6.1, we o er an overview of the LibAFL modules and the respective features that we have used in our implementation. Section 6.2 presents two versions of SCFuzz using LibAFL, applied on a vulnerable password-checking program, and the respective results. The rst version uses return codes as a pseudo-timing side-channel, similar to what we have presented in Section 5.3.2. The second version uses execution time as side-channel information, calculating the timing side-channel score using the high-score approach, as presented in Section 4.2. None of these versions use an instrumentation mechanism for analyzing the edge coverage. Section 6.3 summarizes the main takeaways from the implementation and analysis process.

6.1 Overview of Implementation

Due to the modular design of LibAFL, the process of creating a fuzzer using LibAFL is an incremental process. Therefore, rst we need to make decisions regarding the modules and features we want to use in our fuzzer. We o er an overview of the selected modules and features in the form of a bullet point list. The selection has been made by following the design process decisions presented in Section 4.5, and by following the short tutorial

which can be found in the LibAFL book [3]. Please note that the most important modules are the Observer, Feedback and Objective Feedback, however we also brie y explain the other modules necessary for creating a fully working fuzzer. Moreover, some of these modules are part of the core concepts explained in Section 3.1.4, and thus we only mention the features we are using in our fuzzer here.

- State: The State is simply a container of the data that changes during fuzzing. In our case, the state contains:
  1. *Random Number Generator*
  2. *In-Memory Corpus*: Stores the test cases to be used for fuzzing in memory.
  3. *On-Disk Corpus*: Stores the test cases that result in a \solution" (as de ned by us using the Feedback module) on the disk.
- EventManager: The EventManager keeps track of certain events that are triggered during the fuzzing process, such as adding a new test case to the Corpus. For our purposes, the *SimpleMonitor* is enough.
- Fuzzer: We also need a Fuzzer instance, which alters the State. One of the actions that is being performed is transferring test cases from the Corpus to the Fuzzer. This is done using a simple *QueueScheduler*, which uses a FIFO queue.
- Executor: For our fuzzer implementation, we can use an *InProcessExecutor*, as we do not require a forking functionality.
- Generator: In this instance, we can use a simple *RandPrintablesGenerator* which generates a string of printable bytes.
- Observer: For our implementations, we use a *StdMapObserver* and a *TimeObserver*. The former makes use of a map to track the covered elements of the program under test, and the latter records the execution time of a program run.
- Feedback: In our case, we make use of the *MaxMapFeedback*, which rates a certain input as interesting if there is a value in the Observer's map that is greater than the maximum value registered so far for the same entry, and our custom *MaxTimeFeeback*, which is explained in Section 6.2.2.
- Objective Feedback: The Objective Feedback is similar to the Feedback module, however its goal is to decide if a certain test case is a \solution". In the case of $\text{SCFuzz}_{\text{LibAFL}}$, we rst make use of a *CrashFeedback*, since in our initial implementations we make use of a white-box approach and thus can change the program under test such that

PASSWORD	MAX BRUTE FORCE TIME
mysec	instantly
mysecure	1 minute, 40 seconds
mysecurepass	1 year, 5 months
mysecurepassword	667090 years, 4 months
mysecurepassword123	11724780267 years, 6 days

Table 6.1: Total time to find the correct password when using a brute force approach. The left column shows the password we are trying to find. Please note that the number in the second column represents amount of time to exhaust every single combination, at a speed of 2071.5 MH(megahashes)/s. Realistically, we should find the password much sooner as you have about 50% chance to encounter the correct password after exhausting just 50% of possible combinations, on average

it crashes when the secret data, e.g. the password, is found by the fuzzer.

- Stages: For our purposes, we use *MutationalStage*, which implements AFL's havoc mutator, and executes the program several times in a row using these mutated inputs.

Finally, the fuzzer requires a harness, or in other words, the program that is going to be fuzzed.

6.2 Fuzzing a Password-Checking Program with a Timing Leak

Following the same reasoning as described in Section 4.1, we make use of a password checking program which contains a timing vulnerability. For ease of implementation, this time we will create a Rust version (as LibAFL is written in Rust), since we can simply hard-code the harness into the fuzzer itself. However, with some additional wrappers we can fuzz programs in any programming language. The Rust version of the C password checking program shown in Code 4.4 can be seen in Code 6.1.

```rust
const CORRECT_PASSWORD: &str = "mysecurepassword123"; // Change this to your desired password

fn check_password(entered_password: &str) -> isize {
    let correct_password_chars = CORRECT_PASSWORD.chars();
    let entered_password_chars = entered_password.chars();

    if entered_password.len() != CORRECT_PASSWORD.len() {
        return -1;
    }

    for (index, (c1, c2)) in correct_password_chars.zip(entered_password_chars).enumerate() {
        if c1 != c2 {
            return index as isize;
        }
    }

    return 100;
}

fn main() {
    println!("Please enter your password:");

    let mut password = String::new();
    std::io::stdin()
        .read_line(&mut password)
        .expect("Failed to read line");

    let result = check_password(&password.trim());

    if result >= 100 {
        panic!("Correct password was {}", password);
    }
}
```

Code 6.1: Password Checker with Timing Leak - Rust - Version 1

6.2.1 First Attempt: Response Codes as a Pseudo-Timing Side-Channel using a High-Score Approach

Similarly to SCFuzz$_{AFL}$, we are initially assuming a white-box approach. Our first attempt at implementing SCFuzz$_{LibAFL}$ makes use of response codes to guide the fuzzing process into finding the correct password, with the highest value for a response code, e.g., value 100, representing that the correct password has been given as input, as explained in Section 5.3.2. Moreover, the program crashes, or panics, when the correct password is found, such that this input can be easily recorded by the fuzzer. No instrumentation for analyzing edge-coverage is used in this implementation. As explained before, in this approach we can interpret the response codes as being a pseudo-timing side-channel. The rest of the design process decisions are the same as presented in Section 4.5.

Recording the response code values is done by manually creating a small shared map, which is used by the *StdMapObserver*, such that one position in the map is attributed to one correct character in the password. The map is initialized with zeros. When an input with that character is found, the value at that position is changed to 1. Using the *MaxMapFeedback*, we are trying to maximize the number of consecutive positions in the map that have a value of 1. We illustrate this behavior in Figure 6.1. This version of SCFuzz$_{LibAFL}$ can be found in Appendix B. The system setup is given in Table 4.1 and the results of this implementation can be found in Table 6.2, which shows the amount of time it took the fuzzer to find the correct password.

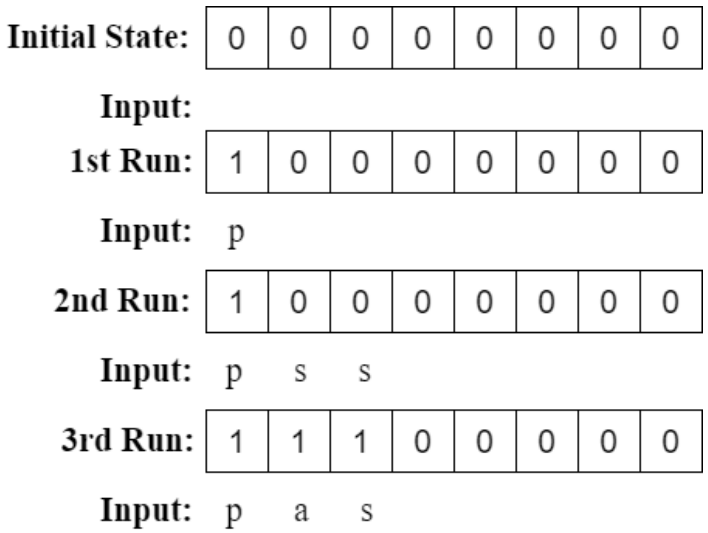

Figure 6.1: The modifications done to the shared map when using the *MaxMapFeedback* module for the correct password *password*

As can be seen, using response codes as a pseudo-timing side-channel is successful, as SCFuzz$_{LibAFL}$ is able to find the correct password in 26 out of 30 fuzzing runs, with variable password lengths and using both letters and digits. Moreover, a longer password seems to result in a longer fuzzing time in most cases. Compared to the baseline results presented in Section 4.4.1, we can see that, in general, this version of SCFuzz$_{LibAFL}$ is more efficient than the unmodified version of AFL - as it can find passwords that are much longer, offers more concrete results, and whether or not an attempt is successful is not influenced by the types of characters used in the password. Finally, we can also see that, in comparison to a brute force approach, this version of SCFuzz$_{LibAFL}$ is more efficient in finding the correct password, especially for higher lengths, as seen in Table 6.2 and Table 6.1. Finally, this version of SCFuzz$_{LibAFL}$ is not actively fuzzing towards increasing edge-

PASSWORD	ATTEMPT #1 TIME	ATTEMPT #2 TIME	ATTEMPT #3 TIME	ATTEMPT #4 TIME	ATTEMPT #5 TIME	AVERAGE TIME
mysec	1 min 13 sec	2 min 1 sec	59 sec	1 min 47 sec	2 min 13 sec	1 min 38 sec
mysecure	2 min 36 sec	3 min 57 sec	4 min	1 min 59 sec	3 min 20 sec	3 min 10 sec
mysecurepass	20 min 46 sec	1 h (!)	10 min 22 sec	12 min 36 sec	13 min 1 sec	14 min 11 sec
mysecurepassword	5 min 45 sec	19 min 15 sec	1 h (!)	25 min 47 sec	29 min 15 sec	20 min
mysecurepassword123	40 min 21 sec	1 h (!)	1 h (!)	50 min 21 sec	49 min 5 sec	46 min 35 sec

Table 6.2: Total time to find the correct password when using $\text{SCFuzz}_{\text{LibAFL}}$ with response codes on the vulnerable password checking program. The first column shows the password we are trying to find, and the other six columns show how long it took $\text{SCFuzz}_{\text{LibAFL}}$ to find that password as well as the average time for the five attempts, respectively. The symbol (!) means that the fuzzing attempt was unsuccessful in finding the correct password, and the average time does not include the unsuccessful attempts

coverage, but it achieves this as a side-effect of finding the correct password, e.g., because of how we have programmed the password checker, finding the correct password also means that complete edge-coverage is achieved, as was explained in more details for the C version in Section 4.4.

6.2.2 Second Attempt: Execution Time as a Side-Channel using a High-Score Approach

For our second attempt at implementing $\text{SCFuzz}_{\text{LibAFL}}$, we want to use the actual execution time of the target as side-channel information, instead of the response codes. Similarly to the first attempt, we are not using instrumentation for analyzing edge-coverage, and the design process decisions are the same as presented in Section 4.5.

For this implementation, we are using a *TimeObserver*, which records the execution time of the target. In LibAFL, there is also a *TimeFeedback*, however, it is not capable of marking inputs as interesting since there are various ways in which the execution time of a program can be interesting. Therefore, we have created our own *MaxTimeFeedback*, which simply implements the high score approach of our timing side-channel score from Section 4.2.

PASSWORD	LENGTH	EXECUTION TIME
pas (correct)	3	3000 ms
pa	2	2000 ms
pabcdefg	8	2180 ms
pabcdefghijkl	8	2370 ms

Table 6.3: Behavior of SCFuzz$_{\text{LibAFL}}$ with regard to execution time when using the high-score approach on the Rust password checking program shown in Code 6.1. Please note that this is an illustrative example, and not the actual results taken during a fuzzing run of SCFuzz$_{\text{LibAFL}}$.

The changes are minimal, as we simply need to associate the high-score with the input that caused the high score, and LibAFL will know to keep this input for further fuzzing. The custom *MaxTimeFeedback* module is shown in Appendix B.

The idea behind this high-score approach is to simulate, and thus replace, the maximization of a shared map using response codes, as we presented in Section 6.2.1. Therefore, our expectations were that this version would result in the same behavior, or even a more e cient and ned-grained way to nd the correct password, with stronger correlations between password length and fuzzing time. However, this was not the case. In fact, this implementation performs worse than the unmodi ed version of AFL or the version of SCFuzz$_{\text{LibAFL}}$ using return codes, as it could only nd passwords up to 4 characters and only containing literals. And even in this case, the fuzzing time required is higher than 45 minutes.

We have further tried to optimize the fuzzing by changing the precision with which the execution time is measured, attempting fuzzing runs using seconds, milliseconds, microseconds and nanoseconds for the measuring. Moreover, we also attempted to force the fuzzer to only try inputs that contain ASCII characters, expecting that this will improve the results, but this was also unsuccessful.

After some analysis, we discovered that during the mutation process, SCFuzz $_{\text{LibAFL}}$ still attempts passwords (lengths) at random, even after forcing the fuzzer to only use inputs that contain ASCII characters, or after using the rst (few) characters of the correct password for the initial input generation. For the rst version of the Rust password checking program speci cally, as shown in Code 6.1, it seems that computing the password length on line 7 causes the execution time to vary enough to cause a new execution time value, regardless of the precision being used. This means that, even if the correct password is only 3 characters long, if SCFuzz$_{\text{LibAFL}}$

tries a 20 characters password or longer, this will result in a new execution time, which is shorter than the execution time for the correct password. This confuses the fuzzer and causes it to fuzz wrong inputs early in the fuzzing process. We illustrate this with the example in Table 6.3, where an execution of the password checking program with the correct password takes 3000 ms, however an execution with a longer, and wrong, password takes less time. Thus, since we use the high-score approach, it is more likely that the fuzzer will keep mutating the longer inputs, making the input with the correct password much harder to reach.

In order to mitigate this behavior, we had to make some changes to the Rust vulnerable password checking program, in order to minimize code that is dependent on computing the length of the password being used as input. Moreover, we have added a `sleep` statement for a duration of 1 second after each character, to ensure bigger time differences if a character check turns out sucessful, similarly to what we have done in Section 5.3.2. The second version of the Rust vulnerable password checking program is seen in Code 6.2.

```rust
let CORRECT_PASSWORD = b"mysecurepassword123"; // Change this to your desired
↪  password

fn check_password(entered_password: &[u8]) -> isize {
    let mut i = 0;

    for _ in buf.iter().zip(CORRECT_PASSWORD).take_while(|(b, c)| b == c) {
        thread::sleep(Duration::from_millis(1000));
        i += 1;
    }

    if i == CORRECT_PASSWORD.len() {
        panic!("Correct password was {}", password);
    }
}
```

Code 6.2: Password Checker with Timing Leak - Rust - Version 2

In addition, with regards to the code of $\text{SCFuzz}_{\text{LibAFL}}$ itself, we have kept the changes that force the fuzzer to only generate inputs that contain ASCII characters, as well as add a fixed threshold for the execution time. Thus, only those new execution times that exceed `current_high_score + threshold` are going to be added to the queue by the fuzzer. This threshold approach was also attempted and explained in Section 5.3.2. These changes to $\text{SCFuzz}_{\text{LibAFL}}$ can be seen in Appendix B. The system setup is given in Table 4.1 and the results of this implementation can be found in Table 6.4, which shows the amount of time it took the fuzzer to find the correct password.

PASSWORD	ATTEMPT #1 TIME	ATTEMPT #2 TIME	ATTEMPT #3 TIME
my	1 min 2 sec	45 sec	1 min 57 sec
mys	3 min 45 sec	1 min 8 sec	2 min 59 sec
myse	30 min (!)	15 min 23 sec	30 min (!)
mysec	5 min 4 sec	30 min (!)	30 min (!)
mysecu	30 min (!)	9 min 37 sec	30 min (!)
mysecu1	30 min (!)	30 min (!)	29 min 28 sec

ATTEMPT #4 TIME	ATTEMPT #5 TIME	AVERAGE TIME
30 sec	52 sec	1 min 1 sec
3 min 2 sec	2 min 20 sec	2 min 12 sec
10 min 34 sec	17 min 43 sec	14 min 33 sec
20 min 49 sec	19 min 7 sec	15 min
25 min 50 sec	30 min (!)	17 min 43 sec
30 min (!)	30 min (!)	29 min 28 sec

Table 6.4: Total time to find the correct password when using $\text{SCFuzz}_{\text{LibAFL}}$ with execution time as a side-channel on the vulnerable password checking program. The first column shows the password we are trying to find, and the other six columns show how long it took $\text{SCFuzz}_{\text{LibAFL}}$ to find that password as well as the average time for the five attempts, respectively. The symbol (!) means that the fuzzing attempt was unsuccessful in finding the correct password, and the average time does not include the unusccesfull attempts

As can be seen, using the execution-time as a side-channel, together with the changes we have made for fine-tuning our implementation, proves to be somewhat successful. This version of $\text{SCFuzz}_{\text{LibAFL}}$ was able to find the correct password in 19 out of 30 fuzzing runs, with variable password lengths and using both letters and digits. However, similarly to the baseline results shown in Section 4.4.1, the execution times vary quite a lot between fuzzing runs, and longer passwords do not necessarily take longer to be found. At the same time, the average times for all passwords seem to be better correlated to the password length. Moreover, we can see that, in general, this version of $\text{SCFuzz}_{\text{LibAFL}}$ is not more efficient than the unmodified version of AFL (as seen in Table 4.2), but does not seem to be influenced by the types of characters used in the password as it was able to find the password `mysecu1` during fuzzing attempt 3. Compared to the results obtained $\text{SCFuzz}_{\text{LibAFL}}$ with return codes, as seen in Table 6.2, this version is again not more efficient, as it also takes a much longer time to find passwords shorter than what $\text{SCFuzz}_{\text{LibAFL}}$ with return codes was able to find. Finally, similarly as before, this version of $\text{SCFuzz}_{\text{LibAFL}}$ is not actively fuzzing towards increasing edge-coverage, but it achieves this as a side-effect of finding the correct password.

6.3 Conclusions

The first conclusion that we can derive from our work on creating SCFuzz$_{\text{LibAFL}}$ is that, indeed, LibAFL is much more straightforward to use and more flexible with regards to the functionality that can be carried out. Moreover, since it also allows fuzzing of not just Rust, but also other programming languages, we believe LibAFL should be one of the go-to options in fuzzing research.

We can also conclude that we were somewhat successful in creating a fuzzer that uses timing side-channel information during the fuzzing process. Our first version of SCFuzz$_{\text{LibAFL}}$, which uses response codes as a pseudo-timing side-channel and a high-score approach for the side-channel score, is able to find the correct password and is generally more efficient in doing so than an unmodified version of AFL or a brute-force approach. However, it does not make any attempts at code coverage, and it cannot be applied in many real world scenarios, as this version requires a lot of fine tuning of the target for the fuzzing to be successful.

Finally, we can also conclude that we were somewhat successful in creating the second version of SCFuzz$_{\text{LibAFL}}$, which uses execution time as side-channel information and a high-score approach for the side-channel score, even though the results are less clear. This version is able to find the correct password in some cases, and in general longer passwords do take longer to be discovered, however there are fuzzing attempts where SCFuzz$_{\text{LibAFL}}$ was not able to find the correct password after 30 minutes or more. The results also do not confirm that our implementation is necessarily more efficient than an unmodified version of AFL or a brute force approach. Finally, this version does not make any attempts at code coverage either, and it can only be applied on a small subset of programs, as it assumes that the interesting outcome is related to longer execution times, and because it requires a lot of fine tuning of the target for the fuzzing to be successful.

Chapter 7

Future Work

Throughout this thesis, we have presented various attempts at creating a fuzzer that uses side-channel information to guide the fuzzing process toward finding the secret data of a program or toward achieving edge-coverage. Due to implementation difficulties and time constraints, we have only focused on creating a fuzzer that uses execution time as side-channel information, a high-score approach for computing the side-channel score and which does not attempt to use this side-channel information for improving edge-coverage. Therefore, there is still work to be performed on this topic.

Firstly, as we have presented in Section 4.2, we have also thought of a second approach for computing the side-channel score, which is the unique descendants approach. If this approach can be successfully implemented, it might allow for the fuzzer to be more generalizible, e.g., allow for the fuzzer to be applied on programs that do not fit the assumption that a higher execution time results in interesting behavior. Moreover, it might solve some of the issues we have encountered with our current implementation, such as the fact that wrong passwords which are longer than the correct password can confuse the fuzzer, and also allow the fuzzer to use side-channel information for edge-coverage.

Secondly, as we have previously mentioned in Chapter 1, one of our original goals was to apply the final version of $\text{SCFuzz}_{\text{LibAFL}}$ on smart card devices. As smart card devices do not allow for instrumentation to be inserted in the code running on them, most of the well-known fuzzer cannot be used. Thus, we believe applying a fuzzer that uses side-channel information during the fuzzing process on such devices is an interesting topic to research further, since it could potentially allow for a black-box analysis of commercial implementations of smart cards applets, for example.

Finally, throughout this thesis, we have only focused on implementing execution time of a program as side-channel information during the fuzzing process. However, there are other side-channels that could potentially be used and could o er interesting results. For example, if power consumption can be used successfully in the fuzzing process, it could o er interesting results especially when used on hardware devices, such as smart cards.

Chapter 8

Conclusion

In our thesis we have presented several attempts at creating a fuzzer, which we call SCFuzz, that aims to use timing side-channel information for guiding the fuzzing process toward improving code coverage and toward finding the secret data, such as the correct password for a password checking program. We described the main decisions made in the design process, the program we have used to test the fuzzer, and the side-channel information being used. Moreover, we have detailed the different versions of SCFuzz and their respective results.

The first implementation we have explained in Chapter 5, $\text{SCFuzz}_{\text{AFL}}$, consists of AFL as the underlying fuzzer and uses its instrumentation that analyzes edge-coverage, in addition to the changes for integrating timing side-channel information. This attempt was unsuccessful, as $\text{SCFuzz}_{\text{AFL}}$ was not able to find the correct password used by a password checking program with timing leakage. We concluded that this is most likely the case because of the random mutations which AFL applies to the inputs.

The second implementation we have explained in Chapter 6, $\text{SCFuzz}_{\text{LibAFL}}$ makes use of the fuzzing library LibAFL, and combines various modules into a working fuzzer, and does not use instrumentation for edge-coverage. The first conclusion regarding this implementation is that LibAFL is a better option for creating a new fuzzer, as it is better documented, much easier to use, and more flexible with regards to the functionality that can be implemented. Secondly, we were somewhat successful in creating two versions of $\text{SCFuzz}_{\text{LibAFL}}$ that use a (pseudo-)timing side-channel for finding the correct password used by a password checking program with timing leakage. The first version, as described in Section 6.2.1, which uses response codes as a pseudo-timing side-channel and a high-score approach for the side-channel score, was able to find the correct password in most attempts, while also being more efficient than an unmodified version of AFL and a brute-force

approach. The second version, as described in Section 6.2.2, which uses execution time as side-channel information and a high-score approach for the side-channel score, approach explained in Section 4.2, was able to find the correct password in slightly more than half of the fuzzing attempts. However, this applies for shorter passwords than the version using response codes, and the results show that this version is not necessarily more efficient than an unmodified version of AFL or a brute-force approach. Moreover, both versions of $SCFuzz_{LibAFL}$ only work for a small subset of programs, as they required a lot of fine tuning of the target for the fuzzing to be successful, and thus take a white-box approach. These versions also assume that a higher side-channel score results in an interesting outcome, such as finding the correct password. However, it might not always be the case that interesting behavior causes higher execution times. We were not successful in obtaining any clear results regarding the effects on edge-coverage using $SCFuzz_{LibAFL}$.

Finally, while we were not completely successful in using a (timing) side-channel in the fuzzing process, we believe these preliminary results offer a good basis for future work. For example, we mention a second approach that could be used for processing the side-channel score and could offer better results, that we did not attempt to implement during our work. Moreover, there are other side-channels, such as power consumption, that could be used during the fuzzing process. Finally, as was our original goal for this thesis, applying a fuzzer that uses side-channel information on a hardware device, such as smart cards, could also have interesting results.

Bibliography

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with input-to-state correspondence. In *Proceedings 2019 Network and Distributed System Security Symposium*. Internet Society, 2019.

[2] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. A systematic review of fuzzing techniques. *Computers & Security*, 75:118{137, 2018.

[3] Andrea Fioraldi and Maier Dominik. \The LibAFL Fuzzing Library". `https://aflplus.plus/libafl-book/libafl.html`.

[4] Andrea Fioraldi and Maier Dominik. \The LibAFL Fuzzing Library Documentation". `https://docs.rs/libafl/latest/libafl/index.html`.

[5] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. Liba : A framework to build modular and reusable fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1051{1065, New York, NY, USA, 2022. Association for Computing Machinery.

[6] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 50{59, 2017.

[7] Google. \AFL GitHub Repository". `https://github.com/google/AFL`.

[8] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[9] Nate Lawson. \Timing attack in Google Keyczar library". https://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/.

[10] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1), jun 2018.

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

[12] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[13] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32{44, dec 1990.

[14] Gedriaan Mulder. Step aside! a fuzzy trip down side-channel lane. *Master Thesis*, nov 2020.

[15] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. Di uzz: Di erential fuzzing for side-channel analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 176{187, 2019.

[16] Yannic Noller and Saeid Tizpaz-Niari. Qfuzz: Quantitative fuzzing for side channels. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 257{269, New York, NY, USA, 2021. Association for Computing Machinery.

[17] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giu rida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society, 2017.

[18] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. *MemLock: Memory Usage Guided Fuzzing*, page 765{777. Association for Computing Machinery, New York, NY, USA, 2020.

[19] Abubakar Zakari, Sai Peck Lee, Khubaib Amjad Alam, and Rodina Ahmad. Software fault localisation: a systematic mapping study. *IET Software*, 13(1):60{74, 2019.

[20] Michal Zalewski. \AFL Technical Paper". https://lcamtuf.coredump.cx/afl/technical_details.txt.

Appendix A

SCFuzz$_{AFL}$

Please Note: There are a few additional, minor modi cations that are not listed here, but can be found on the GitHub repository of this project.

```
/* SP: Calibration Scores
 * START
 */
/* Total calibration score */
static u64 total_score;
/* Highest score / most costly */
static u64 high_score;
/* Lowest score / least costly */
static u64 low_score = 0-1;
 /* If X% of highscore, put in queue */
#define KEEP_MORE_COSTLY_PERC 100
/* If X% of lowscore, put in queue */
#define KEEP_LESS_COSTLY_PERC 100

/* Highest return code */
static u8 return_code_high_score;

/* Use the old culling method  */
static u8 old_culling = 0;
/* END */
```

Code A.1: Score Global Variables

```
struct queue_entry {

  u8* fname;                      /* File name for the test case      */
  u32 len;                        /* Input length                     */

  u8  cal_failed,                 /* Calibration failed?              */
      trim_done,                  /* Trimmed?                         */
      was_fuzzed,                 /* Had any fuzzing done yet?        */
      passed_det,                 /* Deterministic stages passed?     */
```

```
      has_new_cov,                /* Triggers new coverage?           */
      var_behavior,               /* Variable behavior?               */
      favored,                    /* Currently favored?               */
      fs_redundant;               /* Marked as redundant in the fs?   */

  u32 bitmap_size,                /* Number of bits set in bitmap     */
      exec_cksum;                 /* Checksum of the execution trace  */

  u64 exec_us,                    /* Execution time (us)              */
      handicap,                   /* Number of queue cycles behind    */
      depth;                      /* Path depth                       */

  /* START */
  u64 score;                      /* SP: Score based on cost model    */
  u8 return_code;                 /* SP: Password Checker return code */
  /* END */

  u8* trace_mini;                 /* Trace bytes, if kept             */
  u32 tc_ref;                     /* Trace bytes ref count            */

  struct queue_entry *next,       /* Next element, if any             */
                     *next_100;   /* 100 elements ahead               */

};
```

Code A.2: Score Queue Variables

```
/* SP: Resource-based Prioritization
 * START
 */
enum {
  /* 00 */ COST_MODEL_STANDARD,
  /* 01 */ COST_MODEL_TIMING,
  /* 02 */ COST_MODEL_POWER
};
int costmodel = COST_MODEL_TIMING;
/* END */

/* SP: Cost-model Optimization Strategy
 * START
 */
enum {
  /* 00 */ MAXIMIZE_COST
};
int coststrategy = MAXIMIZE_COST;
/* END */

/* SP: Password Checker Return Codes
 * START
 */
enum {
  /* 00 */ LENGTH_HIGHER,
  /* 01 */ FILE_ERROR,
```

```
  /* 02 */ LENGTH_MISMATCH,
  /* 03 */ WRONG_CHARACTER,
  /* 04 */ CORRECT_PASSWORD
};
/* END */
```

Code A.3: Additional Fuzzing Options and Variables

```
static void update_bitmap_score(struct queue_entry* q) {

  u32 i;
  /* SP: Use score field
   * u64 fav_factor = q->exec_us * q->len;
   */

  /* For every byte set in trace_bits[], see if there is a previous winner,
     and how it compares to us. */

  for (i = 0; i < MAP_SIZE; i++)

    if (trace_bits[i]) {

       if (top_rated[i]) {

         /* Faster-executing or smaller test cases are favored. */

         /* SP: Scoring based on resource use
         * START
         * if (fav_factor > top_rated[i]->exec_us * top_rated[i]->len)
↪  continue;
         */
         if (q->score <= top_rated[i]->score && q->return_code <=
↪  top_rated[i]->return_code) continue;
         /* END */

         /* Looks like we re going to win. Decrease ref count for the
            previous winner, discard its trace_bits[] if necessary. */

         if (!--top_rated[i]->tc_ref) {
           ck_free(top_rated[i]->trace_mini);
           top_rated[i]->trace_mini = 0;
         }

       }

       /* Insert ourselves as the new winner. */

       top_rated[i] = q;
       q->tc_ref++;

       if (!q->trace_mini) {
         q->trace_mini = ck_alloc(MAP_SIZE >> 3);
         minimize_bits(q->trace_mini, trace_bits);
```

```
      }

      score_changed = 1;

    }

}
```

Code A.4: Modi ed Bitmap Function

```
/* SP: New method of culling the queue */
static void cull_queue(void) {

  struct queue_entry* q;

  u64 top_scorer_1, top_scorer_2, top_scorer_3;
  top_scorer_1 = top_scorer_2 = top_scorer_3 = 0;
  u8 top_return_code = 0;

  if (old_culling) {
    cull_queue_old();
    return;
  }

  if (dumb_mode || !score_changed) return;

  score_changed = 0;
  queued_favored  = 0;
  pending_favored = 0;

  q = queue;

  while (q) {
    q->favored = 0;

    if (q->score > top_scorer_1) {
            top_scorer_3 = top_scorer_2;
            top_scorer_2 = top_scorer_1;
            top_scorer_1 = q->score;
    } else if (q->score > top_scorer_2) {
            top_scorer_3 = top_scorer_2;
            top_scorer_2 = q->score;
    } else if (q->score > top_scorer_3) {
            top_scorer_3 = q->score;
    }

    q = q->next;
  }

  q = queue;

  while(q) {
    if (q->return_code > top_return_code)
```

```
      top_return_code = q->return_code;

    q = q->next;
  }

  q = queue;

  while (q) {
    if (q->score >= top_scorer_3 || q->return_code >= top_return_code) {
      q->favored = 1;
      queued_favored++;
      if (!q->was_fuzzed) pending_favored++;
    }
    q = q->next;
  }
}
```

Code A.5: Modified Queue Culling Function

```
EXP_ST void setup_shm(void) {

  u8* shm_str;

  if (!in_bitmap) memset(virgin_bits, 255, MAP_SIZE);

  memset(virgin_tmout, 255, MAP_SIZE);
  memset(virgin_crash, 255, MAP_SIZE);

  /* SP: Allocate extra 2*8 bytes for resource results + 1 byte for return
  ↪  code results
   * START
   * shm_id = shmget(IPC_PRIVATE, MAP_SIZE, IPC_CREAT | IPC_EXCL | 0600);
   */
  shm_id = shmget(IPC_PRIVATE, MAP_SIZE + 17, IPC_CREAT | IPC_EXCL | 0600);
  /* END */

  if (shm_id < 0) PFATAL("shmget() failed");

  atexit(remove_shm);

  shm_str = alloc_printf("%d", shm_id);

  /* If somebody is asking us to fuzz instrumented binaries in dumb mode,
     we don t want them to detect instrumentation, since we won t be sending
  ↪  fork server commands. This should be replaced with better auto-detection
  ↪  later on, perhaps? */

  if (!dumb_mode) setenv(SHM_ENV_VAR, shm_str, 1);

  ck_free(shm_str);

  trace_bits = shmat(shm_id, NULL, 0);
```

```
  if (!trace_bits) PFATAL("shmat() failed");

}
```

Code A.6: Additional Shared Memory Allocation

```
 /* SP: Variable for storing the return code of a fuzzing run
  * START
  */
 u8 return_code;
 /* END */

.........

 /* SP: Allocate extra 2*8 bytes for resource results + 1 byte for return
 ↪  code results
  * START
  * memset(trace_bits, 0, MAP_SIZE);
  */
 memset(trace_bits, 0, MAP_SIZE + 17);
 /* END */

.........

/* SP: Retrieve return code value
 * START
 */
 return_code = WEXITSTATUS(status);
 memcpy(&trace_bits[MAP_SIZE + 16], &return_code, 1);
 /* END */
```

Code A.7: Modi cations to *run_target()* function

```
 /* SP: Variables for storing the execution time of a fuzzing run
  * START
  */
 u64 start_entry_time, stop_entry_time, entry_execution_time;
 /* END */

.........

 /* SP: Retrieving the start and end time of the fuzzing run
  * START
  */
 start_entry_time = get_cur_time_us();
 fault = run_target(argv, use_tmout);
 stop_entry_time = get_cur_time_us();
 /* END */

 /* SP: Computing the exectuong time of the current input and storing in the
 ↪  shared memory
```

```
 * START
 */
entry_execution_time = stop_entry_time - start_entry_time;
memcpy(&trace_bits[MAP_SIZE], &entry_execution_time, 8);
/* END */

.........

/* SP: Copy resource use here as well
 * START
 */
u64 time, memory;
memcpy(&time, &trace_bits[MAP_SIZE], 8);
memcpy(&memory, &trace_bits[MAP_SIZE + 8], 8);

if (costmodel == COST_MODEL_TIMING) {
    q->score = time;
} else if (costmodel == COST_MODEL_MEMORY) {
    q->score = memory;
} else {
    q->score = 0-(q->exec_us * q->len);
}
total_score += q->score;
if (q->score > high_score)
    high_score = q->score;
/* END */
```

Code A.8: Modi cations to *calibrate_case()* function

```
/* SP: Add suffix for highscore
 * START
 */
if (new_high_score) {
   strcat(ret, ",high_score");
}
/* END */

/* SP: Add suffix for return code highscore
 * START
 */
if (new_return_code_high_score) {
   if (return_code_high_score == LENGTH_HIGHER)
       strcat(ret, ",length_higher");
   else if (return_code_high_score == FILE_ERROR)
       strcat(ret, ",file_error");
   else if (return_code_high_score == LENGTH_MISMATCH)
       strcat(ret, ",length_mismatch");
   else if (return_code_high_score == WRONG_CHARACTER)
       strcat(ret, ",wrong_character");
   else if (return_code_high_score == CORRECT_PASSWORD)
       strcat(ret, ",correct_password");
}
```

```
 /* END */
```

Code A.9: Modi cation to *describe_op()* function

```
/* SP: Check if this input leads to a (much) more costly path
 * START
 */
static u8 more_costly_than_highscore() {
  u64 time, memory;
  memcpy(&time, &trace_bits[MAP_SIZE], 8);
  memcpy(&memory, &trace_bits[MAP_SIZE + 8], 8);

  if (costmodel == COST_MODEL_TIMING && time > ((high_score *
 ↪  KEEP_MORE_COSTLY_PERC)/100)) {
    return 1;
  } else if (costmodel == COST_MODEL_MEMORY && memory > ((high_score *
 ↪  KEEP_MORE_COSTLY_PERC)/100)) {
    return 1;
  }
  return 0;
}
/* END */

/* SP: Set lowscore if needed
 * START
 */
static void set_lowscore_if_needed() {
  u64 time, memory, instr_cost, user_defined_cost;
  memcpy(&time, &trace_bits[MAP_SIZE], 8);
  memcpy(&memory, &trace_bits[MAP_SIZE + 8], 8);

  if (costmodel == COST_MODEL_TIMING && time < low_score) {
    low_score = time;
  } else if (costmodel == COST_MODEL_MEMORY && memory < low_score) {
    low_score = memory;
  }
}
/* END */
```

Code A.10: Score Processing Functions

```
/* SP: True if the found score is a new highscore and
 *     the new return code is higher
 * START
 */
u8 new_high_score = 0;
u8 return_code, new_return_code_high_score = 0;
/* END */

.........
```

```
/* SP: Also keep if KEEP_MORE_COSTLY_PERC is more costly than the current
 ↪  highscore
 * START
 */
 hnb = has_new_bits(virgin_bits);

 if (coststrategy == MAXIMIZE_COST) {
    new_high_score = more_costly_than_highscore();
 } else {
    new_high_score = 0;
 }

 memcpy(&return_code, &trace_bits[MAP_SIZE + 16], 1);
 if (return_code > return_code_high_score)
    new_return_code_high_score = 1;
 else new_return_code_high_score = 0;

 /* if (!(hnb = has_new_bits(virgin_bits))) { */
 if (!hnb && !new_high_score && !new_return_code_high_score) {
    if (crash_mode) total_crashes++;
        return 0;
    }
/* END */

.........

/* SP: New highscore is treated similarly to increased coverage
 * START
 */
 if (hnb == 2 || new_high_score > 0 || new_return_code_high_score > 0) {
 /* if (hnb == 2) { */
    queue_top->has_new_cov = 1;
    queued_with_cov++;
 }
/* END */
```

Code A.11: Modi cations to *save_if_interesting() function*

```
static void maybe_update_plot_file(double bitmap_cvg, double eps) {

  static u32 prev_qp, prev_pf, prev_pnf, prev_ce, prev_md;
  static u64 prev_qc, prev_uc, prev_uh;
  /* SP: Variables for storing previous low score, previous high score nad
 ↪  previous highest return code
   * START
   */
  static u64 prev_low, prev_high;
  static u8 prev_return_code;
  /* END */

  if (prev_qp == queued_paths && prev_pf == pending_favored &&
      prev_pnf == pending_not_fuzzed && prev_ce == current_entry &&
      prev_qc == queue_cycle && prev_uc == unique_crashes &&
```

```
      prev_uh == unique_hangs && prev_md == max_depth &&
      /* SP: Add low score, high score and return code
       * START
       */
      prev_low == low_score && prev_high == high_score
      && prev_return_code == return_code_high_score) return;
      /* END */

  prev_qp  = queued_paths;
  prev_pf  = pending_favored;
  prev_pnf = pending_not_fuzzed;
  prev_ce  = current_entry;
  prev_qc  = queue_cycle;
  prev_uc  = unique_crashes;
  prev_uh  = unique_hangs;
  prev_md  = max_depth;
  /* SP:
   * START
   */
  prev_low = low_score;
  prev_high = high_score;
  prev_return_code = return_code_high_score;
  /* END */

  /* Fields in the file:

     unix_time, cycles_done, cur_path, paths_total, paths_not_fuzzed,
     favored_not_fuzzed, unique_crashes, unique_hangs, max_depth,
     execs_per_sec, low_score, high_score, return_code */

  /* SP:
   * START
   */
  fprintf(plot_file,
          "%llu, %llu, %u, %u, %u, %u, %0.02f%%, %llu, %llu, %u, %0.02f,
↪  %llu, %llu, %u\n",
          get_cur_time() / 1000, queue_cycle - 1, current_entry,
↪  queued_paths,
          pending_not_fuzzed, pending_favored, bitmap_cvg, unique_crashes,
          unique_hangs, max_depth, eps, low_score, high_score,
↪  return_code_high_score); /* ignore errors */
  /* END */

  fflush(plot_file);

}
```

Code A.12: Adding the scores to the fuzzer plot file

```
/* SP: Retrieving the start and end time of the fuzzing run
 * START
 */
start_entry_time = get_cur_time_us();
```

```
fault = run_target(argv, exec_tmout);
stop_entry_time = get_cur_time_us();
/* END */

/* SP: Computing the execution time of the fuzzing run for the current input
 ↪  and storing in the shared memory
 * START
 */
entry_execution_time = stop_entry_time - start_entry_time;
memcpy(&trace_bits[MAP_SIZE], &entry_execution_time, 8);
/* END */
```

Code A.13: Modi cations to *common_fuzz_stu ()* function

```
static u32 calculate_score(struct queue_entry* q) {

  /* SP: Use score field instead
   * START
   */
  u32 avg_score = total_score / total_cal_cycles;
  /* u32 avg_exec_us = total_cal_us / total_cal_cycles; */
  /* END */
  u32 avg_bitmap_size = total_bitmap_size / total_bitmap_entries;
  u32 perf_score = 100;

  /* Adjust score based on execution speed of this path, compared to the
     global average. Multiplier ranges from 0.1x to 3x. Fast inputs are
     less expensive to fuzz, so we re giving them more air time. */

  /* if (q->exec_us * 0.1 > avg_exec_us) perf_score = 10;
  else if (q->exec_us * 0.25 > avg_exec_us) perf_score = 25;
  else if (q->exec_us * 0.5 > avg_exec_us) perf_score = 50;
  else if (q->exec_us * 0.75 > avg_exec_us) perf_score = 75;
  else if (q->exec_us * 4 < avg_exec_us) perf_score = 300;
  else if (q->exec_us * 3 < avg_exec_us) perf_score = 200;
  else if (q->exec_us * 2 < avg_exec_us) perf_score = 150;*/

  /* SP: Score based on score field
   * START
   */
  if (q->score * 0.25 > avg_score) perf_score = 300;
  else if (q->score * 0.33 > avg_score) perf_score = 200;
  else if (q->score * 0.25 > avg_score) perf_score = 150;
  else if (q->score * 1.5 < avg_score) perf_score = 75;
  else if (q->score * 2 < avg_score) perf_score = 50;
  else if (q->score * 4 < avg_score) perf_score = 25;
  else if (q->score * 10 < avg_score) perf_score = 10;
  /* END */

.........
```

Code A.14: Modi cation to *calculate_score()* function

Appendix B

SCFuzz$_{\text{LibAFL}}$

```
#[cfg(windows)]
use std::ptr::write_volatile;
use std::{path::PathBuf, ptr::write};

#[cfg(feature = "tui")]
use libafl::monitors::tui::{ui::TuiUI, TuiMonitor};
#[cfg(not(feature = "tui"))]
use libafl::monitors::SimpleMonitor;
use libafl::{
    corpus::{InMemoryCorpus, OnDiskCorpus},
    events::SimpleEventManager,
    executors::{inprocess::InProcessExecutor, ExitKind},
    feedbacks::{CrashFeedback, MaxMapFeedback},
    fuzzer::{Fuzzer, StdFuzzer},
    generators::RandPrintablesGenerator,
    inputs::{BytesInput, HasTargetBytes},
    mutators::scheduled::{havoc_mutations, StdScheduledMutator},
    observers::StdMapObserver,
    schedulers::QueueScheduler,
    stages::mutational::StdMutationalStage,
    state::StdState,
};
use libafl_bolts::{current_nanos, rands::StdRand, tuples::tuple_list,
↪  AsSlice};

// Coverage map with explicit assignments due to the lack of
↪  instrumentation
static mut SIGNALS: [u8; 200] = [0; 200];
static mut SIGNALS_PTR: *mut u8 = unsafe { SIGNALS.as_mut_ptr() };

/// Assign a signal to the signals map
fn signals_set(idx: usize) {
    unsafe { write(SIGNALS_PTR.add(idx), 1) };
}

fn main() {
    let mut harness = |input: &BytesInput| {
```

```
        let target = input.target_bytes();
        let buf = target.as_slice();

        const CORRECT_PASSWORD: &str = "mysecurepassword123"; // Change this
↪  to your desired password

        fn check_password(entered_password: &str) -> isize {
                    let correct_password_chars = CORRECT_PASSWORD.chars();
                let entered_password_chars = entered_password.chars();

                    if entered_password.len() != CORRECT_PASSWORD.len() {
                            signals_set(0);
                        return -1;
                    }

                    for (index, (c1, c2)) in
↪  correct_password_chars.zip(entered_password_chars).enumerate() {
                        if c1 != c2 {
                                signals_set(index + 1);
                                    return index as isize;
                        }
                    }

                signals_set(100);
                    return 100;
        }

            let password = String::from_utf8_lossy(buf);
            let result = check_password(&password.trim());

            if result >= 100 {
                    panic!("Correct password was {}", password);
            }

        ExitKind::Ok
    };

    // Create an observation channel using the signals map
    let observer = unsafe { StdMapObserver::from_mut_ptr("signals",
↪  SIGNALS_PTR, SIGNALS.len()) };

     // Feedback to rate the interestingness of an input
    let mut feedback = MaxMapFeedback::new(&observer);

    // A feedback to choose if an input is a solution or not
    let mut objective = CrashFeedback::new();

    // create a State from scratch
    let mut state = StdState::new(
        // RNG
        StdRand::with_seed(current_nanos()),
        // Corpus that will be evolved, we keep it in memory for
↪  performance
        InMemoryCorpus::new(),
```

```
        // Corpus in which we store solutions (crashes in this example),
        // on disk so the user can get them after stopping the fuzzer
        OnDiskCorpus::new(PathBuf::from("./crashes")).unwrap(),
        &mut feedback,
        &mut objective,
    )
    .unwrap();

    // The Monitor trait defines how the fuzzer stats are displayed to the
↪   user
    let mon = SimpleMonitor::new(|s| println!("{s}"));

    // The event manager handles the various events generated during the
↪   fuzzing loop
    // such as the notification of the addition of a new item to the corpus
    let mut mgr = SimpleEventManager::new(mon);

    // A queue policy to get testcasess from the corpus
    let scheduler = QueueScheduler::new();

    // A fuzzer with feedbacks and a corpus scheduler
    let mut fuzzer = StdFuzzer::new(scheduler, feedback, objective);

    // Create the executor for an in-process function
    let mut executor = InProcessExecutor::new(&mut harness,
↪   tuple_list!(observer), &mut fuzzer, &mut state, &mut mgr)
        .expect("Failed to create the Executor");

    // Generator of printable bytearrays of max size 32
    let mut generator = RandPrintablesGenerator::new(32);

    // Generate 8 initial inputs
    state
        .generate_initial_inputs(&mut fuzzer, &mut executor, &mut generator,
↪   &mut mgr, 8)
        .expect("Failed to generate the initial corpus");

    // Setup a mutational stage with a basic bytes mutator
    let mutator = StdScheduledMutator::new(havoc_mutations());
    let mut stages = tuple_list!(StdMutationalStage::new(mutator));

    fuzzer
        .fuzz_loop(&mut stages, &mut executor, &mut state, &mut mgr)
        .expect("Error in the fuzzing loop");
}
```

Code B.1: SCFuzz$_{\text{LibAFL}}$ - Return Values Version

```
#[cfg(windows)]
use std::ptr::write_volatile;
use std::{path::PathBuf, ptr::write};

use std::thread;
```

```rust
use std::time::Duration;

#[cfg(feature = "tui")]
use libafl::monitors::tui::{ui::TuiUI, TuiMonitor};
#[cfg(not(feature = "tui"))]
use libafl::monitors::SimpleMonitor;
use libafl::{
    corpus::{InMemoryCorpus, OnDiskCorpus},
    events::SimpleEventManager,
    executors::{inprocess::InProcessExecutor, ExitKind},
    feedbacks::{CrashFeedback, MaxTimeFeedback},
    fuzzer::{Fuzzer, StdFuzzer},
    generators::RandPrintablesGenerator,
    inputs::{BytesInput, HasTargetBytes},
    mutators::{string::{StringCategoryRandMutator,
 ↪  StringSubcategoryRandMutator},
                scheduled:: StdScheduledMutator,
                },
    observers:: TimeObserver,
    schedulers::QueueScheduler,
    stages::{mutational::StdMutationalStage,
             string::StringIdentificationStage,
             },
    state::StdState,
    Evaluator,
};
use libafl_bolts::{current_nanos, rands::StdRand, tuples::tuple_list,
 ↪  AsSlice};

fn main() {
    let mut harness = |input: &BytesInput| {
        let target = input.target_bytes();
        let buf = target.as_slice();

        let CORRECT_PASSWORD = b"mysecurepassword123"; // Change this to
 ↪  your desired password

        let mut i = 0;

        for _ in buf.iter().zip(CORRECT_PASSWORD).take_while(|(b, c)| b ==
 ↪  c) {
            // signals_set(i);
            thread::sleep(Duration::from_millis(1000));
            i += 1;
        }

        if i == CORRECT_PASSWORD.len() {
            let password = String::from_utf8_lossy(buf);
            panic!("Correct password was {}", password);
        }

        ExitKind::Ok
    };
```

```
    // Create an observation channel using the signals map
    let observer = TimeObserver::new("time");

     // Feedback to rate the interestingness of an input
    let mut feedback = MaxTimeFeedback::with_observer(&observer);

    // A feedback to choose if an input is a solution or not
    let mut objective = CrashFeedback::new();

    // create a State from scratch
    let mut state = StdState::new(
        // RNG
        StdRand::with_seed(current_nanos()),
        // Corpus that will be evolved, we keep it in memory for
↪   performance
        InMemoryCorpus::new(),
        // Corpus in which we store solutions (crashes in this example),
        // on disk so the user can get them after stopping the fuzzer
        OnDiskCorpus::new(PathBuf::from("./crashes")).unwrap(),
        &mut feedback,
        &mut objective,
    )
    .unwrap();

    // The Monitor trait defines how the fuzzer stats are displayed to the
↪   user
    let mon = SimpleMonitor::new(|s| println!("{s}"));

    // The event manager handles the various events generated during the
↪   fuzzing loop
    // such as the notification of the addition of a new item to the corpus
    let mut mgr = SimpleEventManager::new(mon);

    // A queue policy to get testcasess from the corpus
    let scheduler = QueueScheduler::new();

    // A fuzzer with feedbacks and a corpus scheduler
    let mut fuzzer = StdFuzzer::new(scheduler, feedback, objective);

    // Create the executor for an in-process function
    let mut executor = InProcessExecutor::new(&mut harness,
↪   tuple_list!(observer), &mut fuzzer, &mut state, &mut mgr)
        .expect("Failed to create the Executor");

    // Generate 8 initial inputs
    fuzzer
        .evaluate_input(
            &mut state,
            &mut executor,
            &mut mgr,
            BytesInput::new(vec![b m ]),
        )
        .unwrap();
```

```
    // Setup a mutational stage with a basic bytes mutator
    let mutator = StdScheduledMutator::new(tuple_list!(
        StringCategoryRandMutator,
        StringSubcategoryRandMutator,
        StringSubcategoryRandMutator
        StringSubcategoryRandMutator,
        StringSubcategoryRandMutator
    ));

    let mut stages = tuple_list!(
        StringIdentificationStage::new(),
        StdMutationalStage::transforming(mutator)
    );

    fuzzer
        .fuzz_loop(&mut stages, &mut executor, &mut state, &mut mgr)
        .expect("Error in the fuzzing loop");
}
```

Code B.2: SCFuzz$_{\text{LibAFL}}$ - Execution Time Version

```
/ SP: MaxTimeFeedback Implementation
#[derive(Serialize, Deserialize, Clone, Debug)]
pub struct MaxTimeFeedback {
    name: String,
    runtime: Duration,
}

impl<S> Feedback<S> for MaxTimeFeedback
where
    S: State,
{
    #[allow(clippy::wrong_self_convention)]
    fn is_interesting<EM, OT>(
        &mut self,
        _state: &mut S,
        _manager: &mut EM,
        input: &S::Input,
        observers: &OT,
        _exit_kind: &ExitKind,
    ) -> Result<bool, Error>
    where
        EM: EventFirer<State = S>,
        OT: ObserversTuple<S>,
    {
        let observer =
↪   observers.match_name::<TimeObserver>(self.name()).unwrap();
        let maybe_runtime = *observer.last_runtime();

        match maybe_runtime {
            Some(observer_runtime) => {
                if observer_runtime.as_millis() > (self.runtime.as_millis()
↪   + Duration::from_millis(900).as_millis()) {
```

```
                    self.runtime = observer_runtime;
                    println!("Runtime: {:?}", self.runtime.as_secs());
                    println!("Input {:?}", input);
                    Ok(true)
                }
                else {
                    Ok(false)
                }
            },
            None => Ok(false),
        }
    }
}

impl Named for MaxTimeFeedback {
    #[inline]
    fn name(&self) -> &str {
        self.name.as_str()
    }
}

impl MaxTimeFeedback {
    #[must_use]
    pub fn new(name: & static str) -> Self {
        Self {
            name: name.to_string(),
            runtime: Duration::from_micros(0),
        }
    }

    #[must_use]
    pub fn with_observer(observer: &TimeObserver) -> Self {
        Self {
            name: observer.name().to_string(),
            runtime: Duration::from_micros(0),
        }
    }
}
```

Code B.3: SCFuzz$_{\text{LibAFL}}$ - MaxTimeFeedback Implementation