Radboud University

Step aside! A fuzzy trip down side-channel lane

Side-channel assisted fuzzing in embedded systems

Gerdriaan Mulder

13 November 2020

Master thesis

Radboud University, Nijmegen
Institute for Computing and Information Sciences

Supervisor
Dr. Ir. Erik Poll
Radboud University, Nijmegen
`erikpoll@cs.ru.nl`

Second reader
Prof. Dr. Lejla Batina
Radboud University, Nijmegen
`lejla@cs.ru.nl`

Abstract

This thesis investigates how to integrate side-channel information of an embedded system into the feedback loop of a fuzzer. Fuzzers, such as AFL, try to find many execution paths of a fuzzing target by mutating input values such that the target's output changes. Side-channels, such as execution time, expose additional information about a running system. We developed a custom communication interface program that operates between AFL and the smartcard, to provide input filtering and report execution results to AFL. In this thesis, we perform experiments using differential fuzzing with DifFuzz, and mutational/evolutionary fuzzing with AFL. We present two case studies: a custom, password checking applet on a Java Card smartcard, and an unprovisioned SIM (Subscriber Identity Module) card. The password checker implements both a timing-sensitive, and a constant-time password checker. First, we find that, for execution distinguishability in differential fuzzing with DifFuzz, the timing side-channel's measurements need to have nanosecond precision. Next, we fuzz the password checking applet with status words as a *pseudo* side-channel in AFL. The password checker reports three status words: length correctness using (1) overall message length and (2) the value of the length parameter in the message, and overall password correctness (3). We show that AFL finds the correct password length (2), but is not able to find the correct password (3) directly. Next, we use status words in AFL to find all possible instructions of a SIM card (instruction set discovery). We show that AFL finds the instruction set in about 30 minutes of fuzzing time, which is twice as slow as a brute-force approach under the same conditions. Finally, we fuzz the password checking applet with timing as side-channel information in AFL, in addition to status words. We show that AFL uses the timing side-channel, because it focuses on the timing-sensitive password checker, and that AFL finds the correct password length (2). However, AFL does not find the correct password (3) due to the password's complexity and the misinterpretation of the overall message length (1).

Contents

Chapter 1

Introduction

The security of embedded systems can be hard to assess due to their restricted nature. Apart from their small physical size, they have limited processing power, memory, storage, and instruction set. Introspection of software running on an embedded system, such as a *smartcard* is more difficult compared to software running on a PC.

Embedded systems vary in architectural design, and may not be designed to protect secrets adequately. One way to find flaws in embedded systems is by analysing their *side-channels*. Some operations take more time to execute than others (*timing* side-channel), while other operations consume more power (*power* side-channel). By closely monitoring such side-channels, or other informational channels, we can learn more about the inner workings of an embedded system. For example, monitoring an informational channel in passports makes it possible to identify the country of origin[27].

Another way to learn about the way embedded systems (or programs in general) work is by *fuzzing*[25] the target. Typically, a program or system (*fuzzing target*) gives predictable results when it processes input it expects. However, when that input gets distorted, the fuzzing target may crash. Crashes reveal previously unknown execution paths and are used by a fuzzer to generate new inputs that may trigger another (unknown) execution path.

This thesis combines these three topics: fuzzing, side-channels, and smartcards. More specifically, we aim to integrate the information from side-channels of a Java Card smartcard into the fuzzer **AFL**. Ideally, the fuzzer finds inputs that trigger new execution paths more efficiently.

We focus on the following research questions:

1. What fuzzers are already using side-channel information?
2. How can we safely interface a Java Card smartcard with the fuzzer **AFL**?
3. How can we provide the fuzzer **AFL** with side-channel information?

As for the structure of this thesis, we provide preliminaries in Chapter 2 on fuzzers, side-channels, and smartcards. Please read this chapter if you are unfamiliar with any of the these topics. In Chapter 3, we provide related work. We present two case studies in Chapter 4: Java Card smartcard with a custom applet, and a **SIM** card. In Chapter 5, we performed experiments on those case studies; they are presented, along with their results in Chapter 5. Finally, we propose future work (Chapter 6), and conclude our work (Chapter 7).

Chapter 2

Preliminaries

In this chapter, we present the necessary background information regarding the topics covered in this thesis. These topics are: fuzzers (Section 2.1), side-channels (Section 2.2), and smartcards (Section 2.3). Below, we enumerate the concepts introduced for each topic. If you are unfamiliar with some of the concepts and/or terms used, you can find more information about them in the referred sections.

For the remainder of this thesis, we assume the reader is familiar with the terms and concepts presented in this chapter.

Fuzzers (Section 2.1) *fuzzing*, fuzzing *target*, fuzzing *input*, {*white*, *gray*, *black*}*-box* fuzzing, {*generational*, *mutational*, *evolutionary*, *differential*} fuzzing, AFL, DifFuzz

Side-channels (Section 2.2) {*timing*, *power*} side-channel, *constant time* execution, *resource consumption*, *code coverage*, *branching operations*,

Smartcards (Section 2.3) *contact(less)* smartcards, smartcard *terminal*, *Answer To Reset* (ATR), *personalization / provisioning*, *ISO7816*, APDU, *status words*, *Java Card*

2.1 Fuzzers

One of the first, or according to anecdotal evidence, *the* first occurrence of the term *fuzzing* dates back to 1988. Barton Miller was logged in on a UNIX machine through a dial-up connection during a thunderstorm. This thunderstorm caused noise on the telephone line that led to random characters being sent, instead of the intended characters. That effect caused common UNIX utilities to crash due to “faulty input”. Miller coined the term **fuzz**, saying that he “wanted a name that would evoke the feeling of random, unstructured data.”[17]. In that time, Miller wrote the following assignment:

```
  Operating System Utility Program Reliability - The Fuzz Generator


The goal of this project is to evaluate the robustness of various UNIX
utility programs, given an unpredictable input stream.  This project has
two parts.  First, you will build a "fuzz" generator.  This is a program
that will output a random character stream.  Second, you will take the
fuzz generator and use it to attack as many UNIX utilities as possible,
with the goal of trying to break them.  For the utilities that break,
you will try to determine what type of input cause the break.
```

Miller and two graduate students systematically tested about 90 utility programs[23]. More than 24 % of the tested utilities would either crash or loop indefinitely on their (random, long) inputs. Among those utilites were the popular editors `vi` and `emacs`, and the c-shell `csh`. Most crashes were caused by incorrect usage of arrays or pointers, such as null pointer dereferences, out-of-bounds array access, and unbounded input. Even though this research has been around for 30 years, both the crash causes and fuzzing techniques are still actual today.

In the following sections, we cover general fuzzing terminology (Section 2.1.1), various types of fuzzing techniques (Section 2.1.2, and the fuzzer AFL (Section 2.1.3) in more detail.

2.1.1 Terminology

In this section, we introduce and explain the terms used in this thesis regarding fuzzing.

The idea behind a *fuzzer* is to test many *inputs* against a *fuzzing target.* In general, we want to know what inputs cause what behaviour (e.g. execution path) in the fuzzing target. Fuzzing targets may be anything that process input. Common examples are text editors, web browsers, and data processing libraries, but the same principles can also be used for embedded systems such as smartcards.

Fuzzers can have various degrees of insight into their fuzzing target. We distinguish three levels of insight, using the commonly used terms *white-box*, *gray-box*, and *black-box*. In white-box fuzzing, the fuzzer has full access to the source code of the fuzzing target. In gray-box fuzzing, the fuzzer can observe the fuzzer target's internals during execution[21]. In black-box fuzzing, the fuzzer can only query the fuzzing target and record its response.

2.1.2 Fuzzing techniques

With regards to techniques used in fuzzing, we can distinguish the following forms: *generational*, *mutational*, *evolutionary*, and *differential* fuzzing. We briefly explain the various methods.

Generational fuzzing uses a predefined (file) format, and generates inputs conforming to that format. Mutational fuzzing uses a few initial (valid) inputs, and mutate these inputs along the way[21]. **AFL** (Section 2.1.3) is an example of a mutational fuzzer.

Evolutionary fuzzing is a subset of mutational fuzzing. It uses genetic algorithms to mutate inputs. Genetic algorithms combine features from 'parent' inputs to generate 'children' inputs. The children inputs are evaluated using some *fitness* function. Only children that pass this fitness test will be used for future input generation and mutation[26].

Differential fuzzing compares the behaviour of a fuzzing target given either different inputs, or given the same input but in different versions of the fuzzing target[24].

2.1.3 AFL

AFL, short for *american fuzzy lop*[1], is a mutational, evolutionary fuzzer that aims at being practical, configuration-less, and fast. It offers a simple textual user interface for runtime inspection of the fuzzing process. The tool is known for finding vulnerabilities in a number of imaging libraries (`libpng`, ImageMagick, `libtiff`), as well as web browsers (Mozilla Firefox, Apple Safari, Internet Explorer), common security software and libraries (`gnutls`, `openssh`, `gnupg`), and many other tools and libraries[1]. In the following paragraphs, we explain how **AFL** works, based on its technical documentation[15].

AFL instruments the source code of the fuzzing target such that it can keep track of how many times a certain branch is hit at runtime. At each branching point, the code given

[1]The nearly-complete "bug-o-rama trophy case" can be found at [1]

in Code 2.1 is inserted.

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

Code 2.1: Instrumentation code inserted by `AFL` at branching points[15]

The array `shared_mem` (also called *shared memory map*) effectively saves tuples of the form `(source branch, destination branch)`. It preserves directionality, such that it can distinguish execution paths with the same start and end branch, but different intermediate branches. For example, an execution path `A -> B -> C -> D -> E -> F` (containing tuples `AB`, `BC`, `CD`, `DE`, `EF`) is different than `A -> B -> C -> E -> D -> F` (containing tuples `AB`, `BC`, `CE`, `ED`, `DF`), even though both paths start at `A` and end in `F`.

`AFL` only saves inputs that add new tuples to the shared memory map, because this means the fuzzer found a previously unseen execution path.

In Figure 2.1, we show a general impression of `AFL`'s shared memory layout. On the left, you see `prev_location`, on the top `cur_location`. Each intersection has a background colour, varying from white to dark gray. This indicates the number of times `AFL` hit the execution path (tuple) from `prev_location` to `cur_location`.

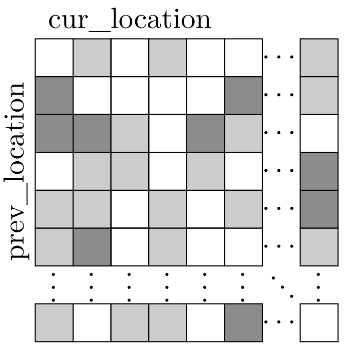

Figure 2.1: Shared memory map in `AFL`

Next to source code instrumentation, `AFL` has a *dumb* and *smart* mode. In the *dumb* mode, `AFL` starts the fuzzing target, feeds it an input, and records the exit code. `AFL` repeats this procedure for each input. Consequently, the fuzzing target needs to be restarted for every input.

In the *smart* mode, `AFL` has more control over the fuzzing target. `AFL` acts like a *command*

and control server: it starts the fuzzing target once, and the fuzzing target waits for commands from `AFL`. The fuzzing target has access to the shared memory map of `AFL` (Figure 2.1) to inform `AFL` about execution paths. This shared memory map is about 64 KB in size.

2.2 Side-channels

Side-channels are observable effects of the execution of a program. For smartcards (Section 2.3) in general, the side-channels *timing*, and *power* are the most relevant. *Timing* side-channels are based on a program's execution time, whereas *power* side-channels are based on a program's power usage. In this thesis, we only cover timing side-channels (Section 2.2.1).

We consider two applications for side-channels:

1. uncovering a secret
2. guiding a fuzzer

Uncovering a secret The first application is based on the program processing secret data, but doing it in such a way that it directly influences the control flow, or that processing is heavy in terms of resource usage. By observing the side-channel, an attacker can reconstruct the secret data[2], as we will see specifically for timing in Section 2.2.1.

Guiding a fuzzer For the second application, we distinguish two approaches:

1. to find two inputs that result in minimum and maximum resource consumption
2. to find enough inputs that result in the biggest code coverage

As an example fuzzing target for these two cases, we consider an input validation function that only processes the complete input when the first character is an `M`.

In the approach using minimum and maximum resource consumption, the fuzzer receives feedback in the form of, for example, the execution time, the number of executed instructions, or power usage of the fuzzing target. The fuzzer tries to find inputs that result

[2]This could be done through something as small as 1 bit, see for example 'Squeezing a key through a carry bit' from the 34th Chaos Communication Congress (2017): `https://fahrplan.events.ccc.de/congress/2017/Fahrplan/events/9021.html`

in lower or higher resource consumption. For example, low resource consumption could indicate that the input is invalid, and that input processing quickly stops. High resource consumption could indicate that additional code paths are visited to fully process the input. Consider the fuzzing target example from the previous paragraph. When looking at resource consumption, every input of any length is discarded equally fast (low resource consumption). However, when the first character matches, it takes longer to process the input (high resource consumption).

In the code coverage approach, the fuzzer receives feedback in the form of, for example, exit codes, or branching information of the fuzzing target. The fuzzer tries to find inputs that result in different exit codes, or in higher number of visited branches. Consider the example fuzzing target shown above. For example, the exit codes could indicate the input is valid or invalid (i.e. does start, or does not start with the character `M`). The branching information could inform the fuzzer that inputs starting with `M` *and* of length larger than 10 characters visit different branches than those having a length smaller than 7 characters.

2.2.1 Timing

Timing side-channels provide information about a program based on the time it takes to execute (a part of) it. As a toy example, we consider two programs `slow` and `fast`, that simply print the supplied arguments. However, `slow` takes 5 seconds to print one argument, and `fast` takes 2 seconds to print one argument. Printing 4 arguments with `slow` takes 20 seconds, whereas this takes 8 seconds for `fast`. Both programs do the same, but they can be distinguished by their runtime: the timing side-channel.

The timing side-channel can be mended by making sure both programs print an argument in equal time. This is called *constant time* execution.

In a more real-world example, we consider the following program that compares an input `PIN`[3] with a stored `PIN`. In Code 2.2, we will identify the timing side-channels present, and in Code 2.3 we will repair the program to make it run in constant time. We assume an attacker can query both functions without constraints.

[3]Personal Identification Number

```go
1  var storedPIN []int{3, 1, 7, 8, 2}
2
3  func CheckPIN(inputPIN []int) bool {
4      if len(inputPIN) != len(storedPIN) {
5          return false
6      }
7      for index, digit := range inputPIN {
8          if storedPIN[index] != digit {
9              return false
10         }
11     }
12     return true
13 }
```

Code 2.2: Vulnerable PIN checker, in *Golang*

In line 4, we see that the input PIN length is compared to the stored PIN length. This on itself is a timing side-channel: when an attacker increases the length of the input PIN, they can derive whether the length was correct. Both a too short and too long PIN would take an equal amount of execution time. The PIN of correct length takes a bit longer to execute, because the program is able to continue.

In line 7–10, the input PIN is compared with the stored PIN, digit by digit. If an input PIN digit does not match with the stored PIN digit at the same index, we see that the function returns `false` directly. This means the execution time of this function is directly related to the number of incorrect PIN digits. An attacker can simply try all possibilities for a single digit, measure the execution time, derive whether a single digit was correct, and then move on to the next digit until they found the stored PIN.

This is troublesome, because we are dealing with secrets. An attacker that is able to observe side-channels, like the ones we've seen above, can derive the stored secret. In general, secrets, or any (derived) part of them, should stay out of comparison operations, such as the `if` statements we have seen in Code 2.2.

In Code 2.3, we see a constant-time implementation for the PIN checker, specifically the part where each input digit is compared to the stored digit.

```go
var storedPIN []int{3, 1, 7, 8, 2}

func CheckPIN(inputPIN []int) bool {
    if len(storedPIN) != len(inputPIN) {
        return false // Still a timing side-channel!
    }
    pinCorrect := 0
    for index := range inputPIN {
        pinCorrect += storedPIN[index] ^ inputPIN[index]
    }
    return pinCorrect == 0
}
```

Code 2.3: Constant-time PIN checker, in *Golang*

The main difference with regards to Code 2.2 is that we process *all* input digits before saying if the PIN was correct. This means an attacker cannot determine through execution time if their first, second, or n-th digit was correct.

We use two operations for this: addition, and the `XOR` operation. In short, the `XOR` operation returns the *difference* of the left and right operand: `a XOR b = 0` when both operands are equal, or 1 when they differ[4]. Consequently, after `XOR`-ing all stored PIN digits with input PIN digits, and adding them, a result of 0 means that all input PIN digits were equal to all stored PIN digits; a non-zero result means at least one digit mismatched.

This approach executes in constant time, because the execution time is not directly related to the number of incorrect input PIN digits.

2.3 Smartcards

Smartcards are embedded systems that offer a wide range of (security-related) applications in a small form factor. Common examples are payment, telephony[5], or access cards. In the following sections, we give background information about smartcards.

First, we give a short overview of smartcard concepts in Section 2.3.1. Then, we look at the standardized data exchange format in smartcards: Application Protocol Data Unit (APDU) in Section 2.3.2. Finally, we look at a specific implementation of a smartcard we use in this thesis, called Java Card in Section 2.3.4.

[4]Another way of looking at `XOR` is using `a XOR b = a + b mod 2`, given that `a` and `b` are binary digits.
[5]Subscriber Identity Module, or SIM

2.3.1 Concepts

In this section we introduce the following concepts regarding smartcards: *ISO7816*, *contact(-less)* smartcards, smartcard *terminal*, *Answer To Reset* (ATR), *personalization / provisioning*.

Smartcards are standardized through various parts of the ISO7816 suite. These standards dictate physical form factor and electrical properties, as well as command and response structures[20].

We distinguish *contact* and *contactless* cards. Contact cards have a recognizable gold-plated section that is divided in several pads. These pads are used for data exchange, and power supply to the card. The smartcard *terminal*[6] is the device that communicates with contact smartcards through these pads. Contactless cards lack such an array of pads. Instead, they use radio waves both as power source and for data exchange.

When inserting a contact smartcard into a smartcard terminal, the smartcard is reset, and the smartcard sends an ATR: *Answer To Reset*. This message informs the smartcard terminal about the smartcard's capabilities and state. Different smartcards can be grouped, or sometimes uniquely identified, by this message[7]. The exact structure of an ATR is out of scope for this thesis. Contactless smartcards have an equivalent message type.

Personalization (or: *provisioning*) is the process of configuring a smartcard application tailored to the user. For example: setting a PIN for a payment card, or attaching a phone number to a SIM card.

2.3.2 Application Protocol Data Unit

An APDU (Application Protocol Data Unit) is the basic structure for ISO7816-4 communication between a smartcard and a smartcard terminal. Seen from a high level point of view, the terminal sends an APDU to the smartcard, the smartcard processes the input, and responds to the command. The structure of the command APDU and response APDU are different. To give an idea of the contents of these APDUs, we show the structure of a command APDU in Table 2.1. The response APDU is more simple and we give its structure in Section 2.3.3.

[6]Another common term is *smartcard reader*

[7]See for example this online ATR parser: `https://smartcard-atr.apdu.fr/`

Code	**Size (bytes)**	**Description**	**Mandatory?**
CLA	1	Class (ISO7816-4, 5.4.1)	Yes
INS	1	Instruction (ISO7816-4, 5.4.2)	Yes
P1	1	First parameter (ISO7816-4, 5.4.3)	Yes
P2	1	Second parameter	Yes
Lc	0, 1, 3	Number of extra command data bytes (1–65535)	No
DATA	Lc	Command data	No, unless Lc is set
Le	0–2	Expected number of response bytes	No

Table 2.1: Structure of a command APDU

2.3.3 Status words

The response APDU contains Le bytes of response data and two bytes containing the status:

- RESP n bytes (n=Le)
- SW1 1 byte
- SW2 1 byte

When we use the term *status words*, we refer to the concatenation of status bytes SW1 and SW2.

ISO7816 defines a 'success' and an 'error' range for status words. Status words of the form 0x9nnn are considered 'success', whereas those of the form 0x6nnn are considered 'error'. For example, 0x9000 means 'no error', and 0x6D00 means 'instruction not supported'.

2.3.4 Java Card

Java Card is a facility to run programs written in Java on embedded systems. It focuses on security applications, interoperability, and provides mechanisms to securely run multiple applets on the same device. Java Card applets will run on any embedded platform that implements the Java Card specification. This makes it possible to write an applet once, and use it on hardware from different vendors. However, it only supports a rather limited subset of Java's functionality. For example, not all data types are supported, nor is garbage collection. Java Card uses the data structures we have seen in Section 2.3.2 and Section 2.3.3 to communicate with the smartcard terminal.

Chapter 3

Related work

In this chapter, we look at publications that are (in)directly related to our thesis' topics: fuzzing, side-channels, and smartcards.

Fuzzing The work of Liang, Pei, Jia, Shen, and Zhang contains[22] a survey of the state of the art in fuzzing. They researched three aspects of fuzzing research: key problems and techniques, usuable fuzzers and their application domain, and future research opportunities. The application domains vary from network protocols, compilers, and general purpose applications. The authors looked at key problems, like scalability and handling crashing test cases, over a number of different fuzzers, with each fuzzer having varying success in each problem domain. The authors like to see the integration of fuzzing as part of the software development cycle.

In another survey paper[21], Li, Zhao, and Zhang review the currently available fuzzing solutions, with a focus on *coverage-based* fuzzing. They show various applications (or targets) for fuzzing: file formats, kernels, and protocols. The paper gives an overview on how to collect seed inputs (to start fuzzing with) and selecting new seeds, what to do for generating test cases (e.g. through machine learning), and how to fuzz applications efficiently.

Side-channels In our thesis, we experimented with DifFuzz [24]. This tool finds side-channel vulnerabilities in Java programs, by comparing the number of executed instructions between different inputs. This is an example of an approach that uses a side-channel to guide the fuzzer (Section 2.2), specifically: the side-channel guides the fuzzer in finding two inputs that result in minimum and maximum resource consumption. Internally, DifFuzz uses fuzzers Kelinci and afl. Their code base is available online, which made it an interesting tool to research. We will see more about DifFuzz in Section 5.1 and Section 5.3.

In [28], Vermoen, Witteman, and Gaydajiev reverse engineered a (further undisclosed type of) Java Card smartcard using power analysis. They recorded each Java Card bytecode's power trace, and devised a way to generate templates of those instructions for later recognition. Using these templates, they could convert a power trace back to Java Card bytecode again. The authors found that the countermeasures against power analysis in their researched Java Card were "not very effective".

Smartcards Aarts, De Ruiter, and Poll performed automatic inference learning on smartcards, specifically EMV bank cards[16]. The authors used bank cards from several banks in The Netherlands, Sweden, Germany, and the UK. Although all cards follow the EMV specifications[4], the authors found identical implementations of the Maestro applications on Dutch bank cards. In general, the learned model of each card was different. We used this idea as inspiration for a discovery experiment (Section 5.5).

Chapter 4

Case studies

4.1 Java Card applet: `PasswordEq`

This thesis focuses on integrating side-channel information of a smartcard in a fuzzer. For experimental purposes, it is useful, if not necessary, to have fine-grained control over the fuzzing target. In our case, this control comprises installing custom software (i.e. applets) on the fuzzing target. Since we can install our own software, we can define what the software does, and how it reacts given some input. This enables us to introduce, amplify or reduce side-channels effects on the fuzzing target.

For this thesis, we created a Java Card applet containing a password checker as fuzzing target. The password checker contains a timing side-channel in one of the operations it can execute. The source code can be found in Section B.1.

The password checker contains a hardcoded password, and two implementations for comparing input with the stored password. One implementation contains a timing side-channel, whereas the other compares the input in constant time (Section 2.2.1). We show in Chapter 5 why: the timing side-channel can be uncovered with very little effort and hardware. Another feature is that we can customize the status words (Section 2.3.3) returned by the applet. In that way, we know what to expect when running the fuzzer against the applet, and thereby verify our approach in using this side-channel as input for the fuzzing process.

In the following paragraphs, we give more details about the implementation.

The applet contains two instructions for checking input with the stored password: safe and unsafe. The safe version uses a constant-time implementation for comparing input with the stored password, that is, it compares all the input bytes up until the length of the stored password, but does not return when one input byte is incorrect. The unsafe version compares the input with the stored password byte-wise. When one byte of input does not match the password at the same position, the function returns. The unsafe version has a timing side-channel.

The applet returns a limited set of status words, at specified points in the execution flow. There are several checks that return *conditions not satisfied*, when P1 and P2 are not `0x00`, *incorrect class* when the first byte is not `0xBA`, *incorrect length* when the data length parameter is not `0x0A` (decimal 10), *password correct* as `0x90FF`, *password incorrect* as `0x66FF`, and of course *instruction not supported* when the instruction byte is not `0x80` or `0x82`.

We use this applet for the following experiments. We show in Section 5.2 that minor/hardly observable timing differences can be observed using standard smartcard communication tools, be it in milliseconds. In Section 5.3, we use an existing fuzzer that optimizes for 'cost' to look for a timing side-channel. When using milliseconds, the distinction is barely visible. However, when using nanosecond precision timing, we can see the difference between the safe and unsafe instructions. In Section 5.4, we use status words returned by the applet as input for vanilla AFL. The idea is that AFL tries to maximize the number of status words observed, because we use the shared memory of AFL and AFL uses that to determine what input hit what code path. Ideally we would want the smartcard applet to be instrumented, but that is difficult if not impossible especially for smartcards in the wild, e.g. banking cards, activated SIM cards, etc. Finally, we use the timing side-channel of the `PasswordEq` applet in Section 5.6 to find out if AFL is able to find the stored password using this timing information.

4.1.1 Implementation details

Now, we give a more thorough description of the Java Card applet `PasswordEq`. This applet contains a static password of 10 bytes (`0x49484746454443424140`) that we assume is safely stored on the smartcard. The applet contains some input checks: it checks the `CLA` byte (`0xBA`), both `P1` and `P2` bytes (`0x00`), and the number of bytes received in total. If the input does not match the expected values, the applet returns `SW_CLA_NOT_SUPPORTED`, `SW_INCORRECT_P1P2`, or `SW_CONDITIONS_NOT_SATISFIED`, respectively. Moreover, we check that the `Lc` byte contains the correct value (`0x0A`) for the length of the data sent. The applet returns `SW_WRONG_LENGTH` if that is not the case.

The applet supports the following instructions:

1. `0x80`: `INS_TRY_PASSWORD_UNSAFE`
2. `0x82`: `INS_TRY_PASSWORD_SAFE`

Both instructions receive a password of 10 bytes as input. Both instructions return either of the following codes:

- `RET_PASSWORD_CORRECT`
- `RET_PASSWORD_INCORRECT`

The first instruction checks each byte of the input with each byte of the stored password. When the check fails at some point (be it the first input byte, or the last), it returns that the password is incorrect. Otherwise, it returns that the password is correct.

The second instruction uses a so-called *constant time* implementation for checking the input against the stored password. In short it executes the following algorithm:

1. Set the variable `passwordEq` to 0
2. `XOR`[1] each input byte with the corresponding password byte
3. Add that result to `passwordEq`
4. Check if the result is non-zero
5. Return the appropriate return code

This method is considered *constant time*, because a) every input byte is processed, regardless of its correctness, and b) the checks are simple, non-branching operations: *addition* and *exclusive or*.

4.2 Unprovisioned simcard

In order to ascertain the validity of using status words as input for a fuzzer, we used the framework built for the PasswordEq applet, and simply substituted the smartcard

[1]In short, the `XOR` operation returns the *difference* of the left and right operand, where 0 means both operands are equal, and 1 means the operands differ.

with an unprovisioned simcard[2]. The idea is that we can match status words with corresponding instructions, perhaps with varying P1/P2 values to uncover non-documented instructions.

In Section 5.5, you can find the experimental setup, and results.

[2]The simcard belonged to a now-defunct company

Chapter 5

Experiments

In this chapter, we present the experiments performed with DIFFUZZ (Section 5.1 and Section 5.3), GlobalPlatformPro tools (Section 5.2), and vanilla AFL (Section 5.4, Section 5.5, and Section 5.6). In the next few paragraphs, we briefly introduce each experiment.

In Section 5.1, we setup the fuzzer DIFFUZZ from [24] as introductory experiment for fuzzing with side-channels. DIFFUZZ uses the side-channel 'number of executed branching instructions'. It runs the same program twice using different inputs, and registers the difference in number of executed branching instructions, calling this metric "delta", denoted by δ. DIFFUZZ tries to maximize δ: a large value for δ indicates the presence of a side-channel, whereas a small value indicates the absence of it.

In Section 5.2, we use the Java Card applet from Section 4.1 and the GlobalPlatformPro tools[5] to intuitively show the purpose-built timing side-channel present in the applet. Although this tool is primarily meant for administrative tasks (installing, removing, locking, etc.) for Java Card smartcards, we found it also prints timing information when sending APDUs.

In Section 5.3, we use the Java Card applet (Section 4.1) with the fuzzer DIFFUZZ in order to find out if DIFFUZZ is suitable for fuzzing a smartcard.

In Section 5.4, we use the fuzzer AFL (Section 2.1.3) and status words (Section 2.3.3) from the Java Card applet (Section 4.1) in order to find the stored password on the smartcard.

In Section 5.5, we use the fuzzer AFL (Section 2.1.3) and status words (Section 2.3.3) to discover the instruction set of an unprovisioned SIM card (Section 4.2).

In Section 5.6, we use the fuzzer AFL (Section 2.1.3), status words (Section 2.3.3) and a timing side-channel (Section 2.2.1) from the Java Card applet (Section 4.1) in order to

Materials used in the experiments The Digital Security department of iCIS provided a dual-boot PC with Ubuntu 16.04.5 LTS[1] to perform experiment 1 (Section 5.1) and 3 (Section 5.3). We used their OmniKey CardMan 5121 smartcard terminal with a JavaCOS A40 Java Card smartcard for experiment 2, 3, 4, and 6 (Section 5.2), Section 5.3, Section 5.4, Section 5.6), and a privately-owned Lenovo Thinkpad x230[2] at the fuzzer side. For experiment 5, we used a privately owned SIM card.

Supporting software In order to execute the experiments with AFL and smartcards (Section 5.4, Section 5.5, Section 5.6), we wrote a *communication interface program* that connects AFL's *smart mode* (Section 2.1.3) with the smartcard, in order to use the smartcard as fuzzing target. AFL sends its inputs to the communication interface program, which in turn performs (optional) filtering, and sends the input as APDU to the smartcard. The metrics we want to capture (e.g. status words, or timing) are written in AFL's shared memory by the communication interface program. In Figure 5.1, you can see the feedback loop from AFL, through the communication interface program and smartcard terminal, to the smartcard, and back to AFL through its shared memory.

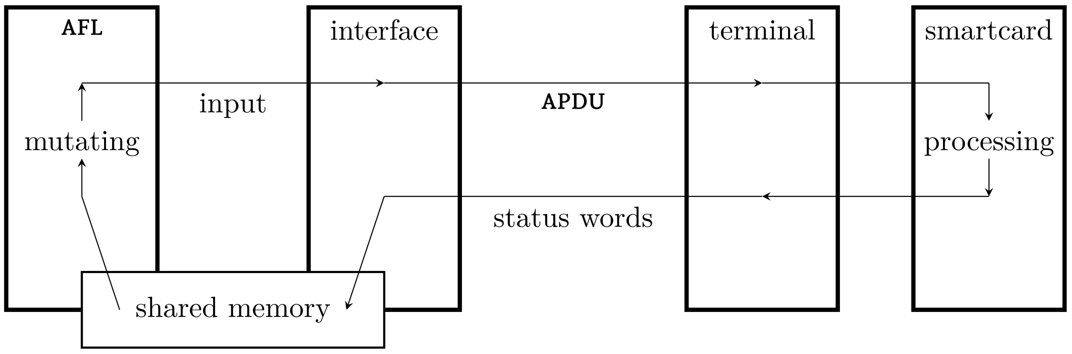

Figure 5.1: Architecture of AFL communicating with a smartcard

[1]Intel i5-4590 @ 3.30 GHz, 32 GB memory + 64 GB swap space, 1 TB disk space
[2]Intel i5-3320M CPU @ 2.60 GHz, 16 GB memory + 8 GB swap space, 250 GB disk space

5.1 Experiment 1: the DifFuzz paper

We re-evaluated 10 test subjects from the DifFuzz paper[24] as stepping stone into fuzzing with side-channels. Their approach is to measure resource consumption in two executions of the same program using two different inputs. The *difference* in number of executed instructions is registered. A large difference is an indicator for the presence of a side-channel vulnerability. The tool tries to find two inputs such that the difference in executed instructions is maximized.

The paper was particularly interesting, because the tool's source code is readily available on GitHub[7]. Their setup contained 8x AMD 8384 (quad-core) @ 2.7 GHz CPUs, 64 GB memory, and OpenSUSE LEAP 42.3. Our setup differs both in CPU and memory specifications, as well as the operating system used: Intel i5-4590 @ 3.3 GHz, 32 GB RAM, Ubuntu 16.04.5 LTS. Nevertheless, the tool was easily up and running in 30-60 minutes, including installing dependencies.

5.1.1 Approach

The source code repository of DifFuzz[7] included an extensive `README` with setup instructions. Nilizadeh, Noller, and Păsăreanu[24] tested their scripts on an Ubuntu 18.04.1 LTS machine, but aside from some missing packages, these scripts also worked fine on our Ubuntu 16.04.5 LTS setup.

We used the *Git tag*[3] `v1.0.0-citable`[4] as basis for our experiments. There are 58 test subjects in the `evaluation` directory. Each test subject contains a Java program and a *driver*. The Java program is the fuzzing target, for example, a password checker. The *driver* connects the fuzzing target to the fuzzer, by passing the input from the fuzzer to the target and recording the instruction cost.

In the original setup[24], every test subject gets 5 runs with a time limit of 30 minutes per run. That adds up to about 6 days of runtime. We chose 5 pairs (safe/unsafe, see next paragraph) of test subjects: *Array*, *unixlogin*, *passwordEq*, *k96*, *apache_ftpserver_md5*.

The *unsafe* version of a test subject contains a side-channel, whereas the *safe* version has this resolved. The results report a metric δ ("delta") that indicates the *difference* in number of executed branching instructions. This number increases when more branching

[3] *Git* is a version control system. A *tag* points to a certain state of a repository

[4] Git commit hash: `2da71abb45f8fa7fd42cb7aa3e2bd4e508287caf`

instructions were observed. When δ approaches 0, little to no branching instructions were observed. In short: a high value of δ indicates the presence of a side-channel, a low value of δ indicates absence of a side-channel.

It should be clear from the tests that the side-channel has been fixed in the *safe* version of the test subject. The pairs we mentioned earlier also appeared in DifFuzz, so we can easily compare results. The estimated runtime was 25 hours.

5.1.2 Results

Our results can be found in Table 5.1. As a reference, the results of the original DifFuzz paper[24] can be found in Table 5.2, where applicable.

Recall that δ ("delta") indicates the *difference* in number of executed branching instructions. The 'Average δ' column shows the averaged value of δ over all executed runs of a test subject: 5 runs per test subject, with a runtime limit of 30 minutes per run. The column 'Standard error' shows the spread of the sampled means. In the column 'Maximum δ', the largest value of δ is reported. Finally, the last column 'Time (s) $\delta > 0$' indicates the time it took, in seconds, since the start of a run that the value of δ became larger than 0.

The dashes indicate that the experiment completed successfully, but no difference in executed instructions were measured. We elaborate on the two cases marked with * in our discussion section below (Section 5.1.3).

Evaluation	**Average δ**	**Standard error**	**Maximum δ**	**Time (s) $\delta > 0$**
Array (safe)	1	0	1	5.8 (± 0.18)
Array (unsafe)	179	7.2	195	6.2 (± 0.44)
unixlogin (safe)	1.8	0	2	207.20 (± 34.16)
unixlogin (unsafe)	2 437 333 341	187 911 943	3 200 000 008	82.40 (± 10.86)
passwordEq (safe)	-	-	-	-
passwordEq (unsafe)	100	1.8	107	9.00 (± 1.10)
*apache_ftpserver_md5 (safe)**	-	-	-	-
apache_ftpserver_md5 (unsafe)	151	0	151	1.00 (± 0.89)
*apache_ftpserver_salted (safe)**	-	-	-	-
k96 (safe)	-	-	-	-
k96 (unsafe)	861 878	388 310	3 695 655	1.80 (± 0.72)

Table 5.1: Results of our DifFuzz evaluation

Evaluation	**Average δ**	**Standard error**	**Maximum δ**	**Time (s) $\delta > 0$**
Array (safe)	1	0	1	7.4 (± 1.21)
Array (unsafe)	192	2.68	195	7.4 (± 0.93)
unixlogin (safe)	3	0	3	510 (± 91.18)
unixlogin (unsafe)	2 880 000 008	286 216 701	3 200 000 008	464.2 (± 64.61)
passwordEq (safe)	0	0	0	-
passwordEq (unsafe)	86.4	20.31	127	8.6 (± 2.11)
apache_ftpserver_md5 (safe)	1	0	1	4.2 (± 1.93)
apache_ftpserver_md5 (unsafe)	151	0	151	2.8 (± 1.11)
apache_ftpserver_salted (safe)	176.4	6.25	198	2.2 (± 0.73)
k96 (safe)	0	0	0	-
k96 (unsafe)	338	185	3 087 339	3.4 (± 0.98)

Table 5.2: Original results from DifFuzz [24]

5.1.3 Discussion

In order to limit the runtime, we chose 5 pairs of test subjects out of the original 58. We did this by copying a provided shell script (`run_evaluation.sh`) to `run_partial_evaluation.sh`, and commenting all but these 5 pairs of test subjects. However, due to the structure of the original shell script, this needed to be done in three different places. Each individual test subject needed a definition in the 'subjects', 'classpaths', and 'drivers' array. Moreover, each array access is done using integer indices, rather than name-based ("associative").

Cases with * We made a mistake in the definition of the 'subjects' array. In the *apache_ftpserver* cases, we used *apache_ftpserver_salted (safe)* where we should have chosen *apache_ftpserver_md5 (safe)*. However, in the 'classpaths' and 'drivers' array, this mistake was absent. As a result, we used the 'Driver_MD5' on the test subject *apache_ftpserver_salted (safe)*, whereas 'Driver_Salted' was meant to be used. Internally, this resulted in an error, but this did not reflect in stopping the fuzzing sessions.

Although we made some minor informational modifications to the shell script (in order to show the number of subjects and estimated runtime before starting the fuzzing session), this error could have been prevented by exiting the fuzzing session at any point where an error occurred. From our point of view, the internals of DifFuzz seemed to work fine.

For future projects using DifFuzz, we recommend to have a closer look at the provided evaluation scripts, and to improve them with better error handling. An error in one of the parts should abort the fuzzing session, and provide the user with feedback on how to solve those errors.

5.1.4 Conclusion

We expected to see little to no values for the difference in executed instructions (δ) in the *safe* versions of the test subjects, because they were supposed to be free of timing side-channel leakage. On the other hand, we did expect values for δ larger than 0 for the *unsafe* versions. The results of our experiments confirm this.

Compared to the results of the original paper[24] (see Table 5.2), our results are similar in most cases. The most notable difference in the results is the *k96 (unsafe)* evaluation, where our *Average* δ and standard error is 200× larger than those from the original paper. We suspect that this is due to experimental setup differences (either in hardware or software), or, given the large discrepancy between average and maximum, that this specific result is incorrectly reported in the paper.

5.2 Experiment 2: Exposing a timing side-channel in a Java Card applet

The GlobalPlatformPro tools[5] make adminstrative tasks around Java Card easily accessible on a regular PC. Apart from installation and removal of applets on Java Card, it provides a way to send customized **APDU**s. Moreover, it provides a timing report for each executed command in its debug mode.

In this experiment, we use this tool to show the presence of a purpose-built timing side-channel in the `PasswordEq` applet from Section 4.1. We use the OmniKey CardMan 5121 as smartcard terminal, and the JavaCOS A40[10] smartcard: a Java Card with 64 KB storage and 1.6 KB of memory. The `PasswordEq` applet has been loaded on this smartcard for this experiment. We ran this experiment on a Lenovo x230, running Arch Linux with the PC/SC toolkit[12], and the aforementioned GlobalPlatformPro tools[5].

5.2.1 Approach

The `PasswordEq` applet (see Section 4.1) contains a hardcoded password, and has two instructions to compare input from a user with that password. These two instructions, however, are different in terms of execution time. One instruction executes the comparison in constant time (see Section 2.2.1), the other instruction executes the comparison character-per-character. Consequentially, the latter instruction's execution time depends on the amount of characters that are match with the hardcoded password.

For the execution of this experiment, we use the GlobalPlatformPro tools[5]. This tool can be used to send arbitrary **APDU**s to the smartcard containing the Java Card applet. Additionally, in debug mode, this tool reports time spent on each command in milliseconds. We use this functionality to expose the timing side-channel present in our applet, `PasswordEq`.

Example In Code 5.1, we give an example of the approach described above. First, we explain how the command (line 1) is constructed. Then, we explain the most important details from the output lines (line 2–5).

[5]For the interested reader: any system that has PC/SC support can be used for this experiment

```
1 $ gp-card --applet 4141414141 --apdu BA8000000A41414141414141414141 -d
2 A>> T=1 (4+0005) 00A40400 05 4141414141           # Select the applet
3 A<<     (0000+2) (22ms) 9000                       # Success
4 A>> T=1 (4+0010) BA800000 0A 41414141414141414141  # Send our password using the unsafe method
5 A<<     (0000+2) (9ms) 66FF                        # RET_PASSWORD_INCORRECT
```

Code 5.1: Debug output of sending an **`APDU`** to the `PasswordEq` applet using `gp-card`

The command construction (line 1) contains the following elements:

- the alias for the GlobalPlatformPro tools executable: `gp-card`;
- the `PasswordEq` applet identifier, hexadecimal encoded: `4141414141`;
- the **`APDU`** we want to send, hexadecimal encoded: `BA8000000A41414141414141414141`[6];
- indication for debug mode (in order to get timing information): `-d`

Each output line (line 2–5) contains the following important elements:

- direction (terminal to smartcard: `>>`; smartcard to terminal: `<<`);
- hexadecimal encoded byte sequences (terminal to smartcard: **`APDU`** sent: `00A4040...` and `BA80...`; smartcard to terminal: status words received: `9000` and `66FF`);
- for status words: the time it took to receive the response in milliseconds (`22ms`, `9ms`)
- manually added comment for in-line clarity (`#`)

As you can see, we can extract the time spent for verifying the password in line 5: `9ms`.

We repeated this command for various inputs (correct/incorrect) and methods (unsafe/safe). The results can be found in Table 5.3

[6]Recall Table 2.1, the dissection of the example **`APDU`**: `CLA`: `BA`, `INS`: `80`, `P1`: `00`, `P2`: `00`, `Lc`: `0A`, `DATA`: `41414141414141414141`

5.2.2 Results

Method	**Password used**	**Correct/incorrect**	**Time spent (ms)**
Unsafe	`41414141414141414141`	Incorrect	9
Unsafe	`49484746454443424140`	Correct	10
Safe	`41414141414141414141`	Incorrect	11
Safe	`49484746454443424140`	Correct	11

Table 5.3: Comparison of execution time of the `passwordEq` applet

We repeated these time measurements a couple of times to ensure that the results are consistent and reproducable across tries.

5.2.3 Discussion

We note that the millisecond precision is somewhat inaccurate, but good enough for illustrative purposes.

It would be interesting to see whether we can see *how many* characters were incorrect. However, given the accuracy of these measurements using the `gp-card` tool, we do not think this is possible with this tool. It might be *made* possible by either modifying the source code of the tool to report nanoseconds, or exaggerate the timing effect in the `PasswordEq` applet itself.

5.2.4 Conclusion

The goal of this experiment was to reveal a purpose-built timing side-channel in the `PasswordEq` applet (Section 4.1) using the GlobalPlatformPro tools[5]. We observed different response times in the *unsafe* case (9 ms vs. 10 ms) but equal response times in the *safe* case: 11 ms.

We conclude that the purpose-built timing side-channel is observable using the GlobelPlatformPro tools.

5.3 Experiment 3: Using DifFuzz to fuzz a Java Card applet on a smartcard

In this experiment, we want to find out if DifFuzz' approach is also suitable for fuzzing Java Card applets on a smartcard. More specifically, we want to know if differential fuzzing also works when provided with timing information, instead of number of executed branching instructions. Ideally, DifFuzz finds the timing side-channel present in our fuzzing target: the `PasswordEq` applet (Section 4.1).

We use timing information measured from the fuzzer's perspective. In this way, there is little to no modification needed on the smartcard itself. It would be interesting future work to use the actual number of executed branching instructions on the smartcard, and we give hints for such an approach in Chapter 6 (Future work).

Compared to number of executed branching instructions, timing information is similar in terms of monotonicity. When a program executes more branching instructions, it also takes more time execute. Therefore, we think that substituting instruction count with timing is suitable for differential fuzzing.

5.3.1 Approach

The approach for this experiment is similar to the one performed in Section 5.1. We use DifFuzz as fuzzer, but instead of fuzzing many Java programs, we only fuzz a single Java Card applet, `PasswordEq` (Section 4.1), on a smartcard. Similar to the fuzzing targets used in Section 5.1, the fuzzing target for this experiment contains a *safe* and an *unsafe* version. The safe version performs a password check in constant time, whereas the unsafe version contains a timing side-channel (see Section 4.1 for applet details).

Instead of counting branching operations as the fuzzing target executes, we register time spent between sending an APDU and receiving the status words.

For this experiment, we used the same PC hardware (Intel i5-4590 @ 3.3 GHz, 32 GB RAM) and operating system (Ubuntu 16.04.5 LTS) used in Section 5.1. We used the same smartcard terminal used in Section 5.2: OmniKey CardMan 5121. The JavaCOS A40 smartcard contained the `PasswordEq` applet. We used the same number of runs and maximum runtime per test subject: 5 runs per test subject, and maximum runtime of 30 minutes.

We ran two fuzzing sessions. Their results can be found in Section 5.3.2. In our first session, the millisecond precision was too low to distinguish the safe and unsafe case. In the second session, we used nanosecond precision, and that proved to be quite effective: we could now distinguish more easily between the safe and unsafe case.

5.3.2 Results

In the next paragraphs, we present the results of this experiment. First, we explain the meaning of each column in the result tables.

DifFuzz generates two inputs that are sent to the `PasswordEq` applet. For each input, we register the time spent between sending the input, and receiving the response (execution time). Then, we use δ ("delta") to indicate the *difference* in execution time between these two inputs.

The 'Average δ' column shows the averaged value of δ over all executed runs of a test subject (5 runs per test subject, with a runtime limit of 30 minutes per run). The column 'Standard error' shows the spread of the sampled means. In the column 'Maximum δ', the largest value of δ is reported. Finally, the last column 'Time (s) $\delta > 0$' indicates the time it took, in seconds, since the start of a run that the value of δ became larger than 0.

(I) using millisecond precision In this paragraph, we show the results of the first fuzzing session.

Evaluation	**Average δ**	**Standard error**	**Maximum δ**	**Time (s) $\delta > 0$**
JavaCard_PasswordEq (safe)	1 000 000	0	1 000 000	24.2 (± 5.35)
JavaCard_PasswordEq (unsafe)	1 000 000	0	1 000 000	13.2 (± 1.95)

Table 5.4: Results of the DifFuzz evaluation, millisecond precision

We note here that our cost measurement was reported in *nanoseconds*. Therefore, the difference in cost, δ, is also in nanoseconds. The average and maximum δ translate to 1 millisecond for both safe and unsafe versions.

Results (II) using nanosecond precision In this paragraph, we show the results of the second fuzzing session.

Evaluation	**Average δ**	**Standard error**	**Maximum δ**	**Time (s) $\delta > 0$**
JavaCard_PasswordEq (safe)	488 499	54 968	623 130	11.40 (± 0.83)
JavaCard_PasswordEq (unsafe)	301 404	37 135	384 562	14.80 (± 2.67)

Table 5.5: Results of the DifFuzz evaluation, nanosecond precision

We note here that our cost measurement was reported in *nanoseconds*. Therefore, the difference in cost, δ, is also in nanoseconds. The average and maximum δ translate to 0.5 milliseconds in the safe case, and 0.3 milliseconds in the unsafe case.

5.3.3 Discussion

In this experiment, we encountered both technical, and fundamental problems. The fundamental problem was the timing precision initially used. The technical problems comprised: *driver* changes not working, logical bug, and sign extension bug in the terminal code.

In the following paragraphs, we elaborate on these problems, and how we solved them.

Timing precision Although we claim the cost measurement was reported in nanoseconds, given the perfectly rounded results in Results (I), this claim seems not entirely accurate. Indeed, if we take a closer look at the documentation behind Java's `time.Instant`[6], we find that this defaults to milliseconds[3].

In order to solve this problem, we switched to `System.nanoTime()`[14] for our timing measurement. However, the documentation made no guarantees on the implemented resolution, which could still be milliseconds[14]. Fortunately, after we adapted the time measurement code to use `system.nanoTime()` instead of `time.Instant`, it became clear that the resolution was actually in nanoseconds. Results (II) (Section 5.3.2) show the effect: we can now distinguish the safe and unsafe cases, in terms of timing.

Driver changes not working In a few preliminary fuzzing sessions, we noticed that changes to the *driver* (the component that connects the fuzzing target to DifFuzz) did not come into effect after compiling. DifFuzz would still use an older version of the driver, instead of the freshly compiled one.

After inspecting the helper scripts more closely, we noticed that an internal component, KELINCI, was still running after we manually stopped the fuzzing session. In the regular flow, the helper scripts would shut down KELINCI, but these scripts did not account for manual interruption. Therefore, changes to the driver would never be used, because the old KELINCI process was still running when restarting the fuzzing session. This was fixed by making sure these processes were killed both in the regular flow, and when manually interrupting the fuzzing session. Changes to the driver would now come into effect after compiling and restarting the fuzzing session.

Logical, and sign extension bug For debugging purposes, we wanted to capture 'unknown status words', i.e. status words that we did not program into the applet. Instead of capturing those, we captured *all* status words. Consequently, our logs were full of 'Unknown status word 0x....'. We could quickly resolve this logical bug.

However, this logical bug actually revealed another bug! We logged all communication from and to the smartcard (APDUs, status words, and the aforementioned debug logging). When summarizing the received status words, we found that *one* APDU resulted into the status words `0x90FF`, or `RET_PASSWORD_CORRECT`. However, upon analysis of the APDU sent, we found that the hexadecimal representation of the sent password was `0x16`, `0x45`, `0x93`, `0x35`, `0x5B`, `0x00`, `0xF2`, `0xFF`, `0xFF`, `0xFF`; nothing close to the expected `0x49`, `0x48`, `0x47`, `0x46`, `0x45`, `0x44`, `0x43`, `0x42`, `0x41`, `0x40` (Section 4.1).

Bytes in Java Card are 8 bits signed integers, this means they range from -128 up to and including 127 decimal. `XOR`ing two bytes yields a byte again. Adding such results may not fit in a byte. Therefore, we first convert (cast) each byte into a *short*: 16 bits signed integers (range: -32 768 up to and including 32 767). However, casting a byte to a short in Java Card *keeps* sign information, which is called *sign extension.* Adding a mix of positive and negative shorts may yield 0, and this was exactly what happened. This was fixed by clearing (*masking*, setting to 0) the upper 8 bits after the cast to short.

5.3.4 Conclusion

In this experiment, we wanted to find out if DIFFUZZ' approach is also suitable for fuzzing Java Card applets on a smartcard. More specifically, we wanted to know if differential fuzzing works when provided with timing information. Ideally, DIFFUZZ found the timing side-channel present on our fuzzing target: the `PasswordEq` applet.

DifFuzz' approach is suitable for fuzzing Java Card applets on a smartcard. Furthermore, differential fuzzing also works when using execution time as metric. We found out that nanosecond precision is better than millisecond precision, because the latter produced indistinguishable results.

DifFuzz found the timing side-channel, given nanosecond timing precision, because the safe (constant-time) and unsafe (timing-sensitive) case can be distinguished.

Finally, we would like to state that fuzzing the Java Card applet actually revealed an unintended side-effect of casting number-like types in Java Card: sign extension.

5.4 Experiment 4: Status words as pseudo side-channel information for AFL

In this experiment, we fuzz a Java Card applet on a smartcard using status words (Section 2.3.3) as a *pseudo* side-channel for fuzzer AFL (Section 2.1.3). The smartcard's applet (Section 4.1) is under our control, and contains a password of fixed length. The applet reports information about the password length, and overall password correctness through status words. We expect that AFL finds the correct password length. Ideally, AFL also finds the stored password based solely on this side-channel.

We use the term *pseudo* side-channel, because status words are part of regular smartcard communication. This also means that they are easily accessible, and do not need additional, specialized hardware to capture. Moreover, if the source of a smartcard's applet is unavailable, status words are—without additional hardware or measurements—the only informational channel in smartcard communication.

In short, we let AFL generate APDUs (Section 2.3.2), send these to the `PasswordEq` applet (Section 4.1) on the smartcard, and feed AFL the returned status words. To speed up fuzzing, we give AFL an example APDU that already contains the correct class, and instruction byte. We expect that AFL automatically finds the correct parameter bytes (P1P2, Section 2.3.2) to interact with this applet. We expect that AFL finds the correct password length based on these status words. Ideally, AFL finds the stored password.

For this experiment, we use non-modified AFL, version 2.95b, our own communication interface program (Section 5) for AFL—smartcard communication, a JavaCOS A40 Java Card smartcard containing the `PasswordEq` applet (Section 4.1), and the OmniKey CardMan 5121 smartcard reader. The overall architecture can be found in Figure 5.1

5.4.1 Approach

We use status words (Section 2.3.3) returned by the smartcard's applet (`PasswordEq`, Section 4.1) as side-channel information for the fuzzer AFL. The fuzzing process works as follows: AFL generates APDUs, the APDU is sent to the smartcard, the `PasswordEq` applet processes the APDU, and returns status words upon completion. The returned status words are registered in AFL's shared memory (Section 2.1.3).

AFL interprets bytes set in shared memory as hit count for a specific branch (Section 2.1.3). Smartcards may return status words at any point in their execution. Therefore, we can see these status words as ‘the smartcard applet hit a specific branch’, and use them to inform **AFL** of the (rough) execution path of the smartcard.

The status words are registered in **AFL**’s shared memory. This works as follows: **AFL**’s shared memory region is 64 KB large[7]. Recall that status words are defined as `SW1` and `SW2` concatenated (Section 2.3.3). We interpret this concatenation as a 16 bit unsigned integer. This limits the value of the status words to `0x0000`–`0xFFFF`[8], or (decimal) `0`–`65 535`. Therefore, we can use the *value* of the status words as (array) index in **AFL**’s shared memory. For example, if the `PasswordEq` applet returns `0x66FF` (“password incorrect”), we increment `shared_memory[0x66FF]`.

We seeded the fuzzer **AFL** with the following byte sequence: `0xBA8000000A4141410A`. The applet itself was already selected on the smartcard through the communication interface program.

5.4.2 Results

We ran a fuzzing session of about 2.5 hours. In this session, **AFL** generated 2 101 613 **APDU**s. Due to filter restrictions, 599 496 **APDU**s were blocked from being sent to the smartcard. The smartcard returned 1 502 118 status words, yielding a global hitrate of 71.5%. On average, we sent just under 129 **APDU**s per second to the smartcard, and received just under 92 status words per second.

The distribution of status words returned from the smartcard:

Status message	**Status words**	**Count**	**Notes**
Wrong P1/P2	`0x6A86`	1 140 708	
Wrong length	`0x6700`	357 285	
Incorrect password	`0x66FF`	3 959	`INS_TRY_PASSWORD_UNSAFE` (`0x80`)
Incorrect password	`0x66FF`	165	`INS_TRY_PASSWORD_SAFE` (`0x82`)
Generic success	`0x9000`	1	
Total		1 502 118	

Table 5.6: Returned status words by the `PasswordEq` applet, sorted on occurrence

We elaborate on the interpretation of the results in Section 5.4.3.

[7]To be precise: an array of bytes of length 65 536

[8]ISO7816 defines `0x6nnn` and `0x9nnn` as standardized ranges for status words[20, Section 5.4.5]

5.4.3 Discussion

Interpretation of the results The one APDU that returned status words `0x9000` is the `SELECT APPLET` command that our communication interface program executes during communication setup. Apart from that, we see that AFL generated 4 124 APDUs that conformed to the applet's expected combination of class, instruction, parameters, and length specification. AFL also found out the *two* instructions that could be used. Although the generated 'passwords' were incorrect (hence the returned status words `0x66FF`), these results indicate that, eventually, the correct password will be generated by AFL, solely based on the information given through status words.

Additional metrics An interesting additional metric would be "time to first 'length correct, instruction correct'". This metric would indicate the time it takes before the fuzzer generated an APDU that contains the correct instruction, and the correct length parameter. In this experiment, this would not be beneficial, because we seeded the fuzzer with a working, yet incomplete, APDU that would have both the correct instruction, and a correct length parameter.

5.4.4 Conclusion

In this experiment, we fuzzed a Java Card applet on a smartcard using status words as *pseudo* side-channel for the fuzzer AFL (Section 2.1.3). The smartcard's applet (Section 4.1) was under our control, and contained a password of fixed length. We expected that AFL finds the correct password length. Ideally, AFL found the stored password based on this pseudo side-channel.

For this experiment, we expected that AFL would autonomously find

1. the correct parameter bytes to interact with the applet, and
2. the correct password length, and
3. the stored password (ideally).

AFL succeeded in (1) and (2), because we registered status words that indicate the password was incorrect. This means that AFL had already found the correct parameter bytes, as well as the correct password length. However, AFL did not find the stored password based on this information.

5.5 Experiment 5: Smartcard instruction set discovery using status words

Status words returned by a smartcard indicate the code path taken given a certain instruction. Given that different code paths return different status words, fuzzing the smartcard using status words as side-channel information should steer the fuzzer into finding more distinct code paths, and thereby increase overall code coverage.

The goal of this experiment is to determine a 'working' instruction set for a smartcard in a black-box approach, using status words as side-channel information for the fuzzer AFL. We consider a combination of class and instruction byte as *working* when the smartcard returns either 'success' status words (`0x9nnn`, see Section 2.3.3), or 'error' status words (`0x6nnn`) except those that indicate that the class, instruction, or parameters were incorrect.

We expect that this approach performs faster than a brute-force one, because the fuzzer is in a feedback loop with the fuzzing target. This should eliminate non-interesting inputs quickly. In order to compare performance, we assume a brute-force approach has access to the same information, and is able to make decisions based on the returned status words.

The fuzzing target is a non-personalized SIM card[9]. We consider this a black-box approach, because we do not have access to the source code of the SIM card. We do not exactly know the implemented instruction set, nor the possible status words. Even though SIM cards are standardized through, e.g. GSM 11.11[19] and various other standards, this still leaves room for undocumented instructions and status words.

For this experiment, we use a non-personalized SIM card as fuzzing target, non-modified AFL, version 2.95b, our own communication interface program (Appendix B.3) for AFL—smartcard communication, and the OmniKey CardMan 5121 smartcard terminal.

5.5.1 Approach

The approach for this experiment is equal to the approach in Section 5.4.1: we use status words (Section 2.3.3) returned by the SIM card as side-channel information for the fuzzer AFL. The fuzzing process works as follows: AFL generates APDUs, the APDU is sent to the

[9]Although the SIM card is tied to specific telecom operator, which is not active on the market anymore, personalizing here means assigning a phone number to it

SIM card, the SIM card processes the APDU, and returns status words upon completion. The returned status words are registered in AFL's shared memory (Section 2.1.3).

We used the byte sequence (hexadecimal encoded) `0xBA8000000a4141410A` as seed input for AFL. Apart from generic safeguards detailed in the next paragraph, we did not setup any filters that would allow or block specific prefixes for APDUs (Section 5).

As a safeguard, we discard generated APDUs that are smaller than two bytes, or larger than 261 bytes. The former condition is used to guarantee sending at least the instruction byte, otherwise we expect that the card always replies with 'instruction not supported'. The latter condition is due to restrictions on the smartcard terminal side, and consequently the maximum message length that we can sent[10].

5.5.2 Results

We executed a fuzzing session of about 30 minutes. In this session, AFL generated 33 280 APDUs that could be sent to the smartcard. This does not include the discarded APDUs that were smaller than two bytes, or larger than 261 bytes, due to restrictions mentioned in Section 5.5.1. We received 26 421 status words from the smartcard, yielding a global hitrate of 79.4%. On average, we could send about 17 APDUs per second.

First, we show the occurrence of the received status words, and those that were rejected before being sent, in paragraph *Status words* and Table 5.7. Then, in paragraph *Instruction set*, we further filter the results, and count the distinct combinations of class, instruction, parameters, and status words (Table 5.8). Next, we report the 109 distinct combinations of class and instruction in Table 5.9. Finally, we make a comparison between this approach, and a brute-force one.

Status words The returned status words can be divided into 'success' (`0x9nnn`) and 'error' (`0x6nnn`). We registered 20 distinct status words, of which 4 fall in the 'success' range, and 16 in the 'error' range. The overall distribution can be found in Table 5.7. In 6 859 cases, the generated APDU could not be sent to the smartcard ('Unknown error'); this figure is shown separately in Table 5.7. Note that 17 cases with the same error message *did* come from the smartcard. We used the GSM 11.11[19] standard to interpret

[10]The relevant information can be found in the smartcard terminal's property list, specifically `dwMaxCCIDMessageLength`. For the OmniKey CardMan 5121, this is 261 bytes, excluding the header: `https://ccid.apdu.fr/ccid/readers/CardMan5121.txt`. More background information can be found at `https://ludovicrousseau.blogspot.com/2014/11/ccid-descriptor-statistics.html`

the returned status words (**Notes** column). We did not encounter undefined or custom status words.

In the **Status message** column, we emphasized three distinct messages: *No EF selected*, *No error*, and *Proactive* SIM. We briefly elaborate on these messages in the Discussion, Section 5.5.3.

Status message	**Status words**	**Count**	**Notes**
Class not supported	`0x6E00`	18 410	-
Instruction not supported	`0x6D00`	3 791	-
Wrong P1/P2	`0x6B00`	3 211	-
No EF selected	`0x9400`	390	GSM 11.11[19, 9.4.4]
Wrong length[11]	`0x6708`	149	Length should be `0x08`
Secure messaging not supported	`0x6882`	126	-
Wrong length[11]	`0x676E`	66	Length should be `0x6E`
No error	`0x9000`	63	-
Proactive SIM	`0x9138`	34	GSM 11.11[19, 9.4.1]
Wrong length[11]	`0x6710`	28	Length should be `0x10`
Logical channel not supported	`0x6881`	23	-
Wrong length[11]	`0x6702`	22	Length should be `0x02`
Wrong length[11]	`0x6700`	22	Correct length unspecified
Unknown error	`0x6F00`	17	Returned by smartcard
Wrong length[11]	`0x6703`	15	Length should be `0x03`
Function not supported	`0x6A81`	14	-
Wrong length[11]	`0x671A`	14	Length should be `0x1A`
Wrong data	`0x6A80`	13	-
Wrong length[11]	`0x6714`	12	Length should be `0x14`
Proactive SIM	`0x9111`	1	GSM 11.11[19, 9.4.1]
Subtotal		26 421	
Unknown error	`0x6F00`	6 859	Did not reach smartcard
Total		33 280	

Table 5.7: Returned status words by the SIM card, sorted on occurrence

[11]SW2 indicates the expected length[19, 9.4.6], or `0x00` when "no additional information is given"; the standard does not clearly specify this last condition

Instruction set We filtered the raw results to exclude *Class not supported*, *Instruction not supported*, *Incorrect P1/P2*, and *Unknown error* if it did not come from the smartcard. This gives us 1 009 status words. The number of distinct combinations of class, instruction, parameters, and returned status words is 539. The exact division is shown in Table 5.8. You can see we found 55 distinct classes, further specified into 109 distinct combinations of class and instruction combination, and so on.

Class	**Instruction**	**P1**	**P2**	**SW1**	**SW2**
55	▨	▨	▨	▨	▨
109		▨	▨	▨	▨
235			▨	▨	▨
522				▨	▨
533					▨
539					

Table 5.8: Distinct combinations of class, instruction, parameters, and status words

To give an idea of the distribution of class and instruction combinations, you can find those grouped by class in Table 5.9.

Class	Instruction
0x00	0x00
	0x70
0x3a	0x80
0x45	0x7f
	0x80
0x4a	0x80
0x5b	0x00
0x7a	0x80
0x86	0x80
0x8a	0x80
0x8d	0xed
0x9a	0x80
	0x87
	0x9f
0xa0	0x04
	0x10
	0x12
	0x14
	0x20
	0x24
	0x26
	0x28
	0x2c
	0x32
	0x44
	0x88
	0x98
	0xa1
0xa0	0xa2
	0xa4
	0xb0
	0xc0
	0xc2
	0xd6
	0xe0
	0xe4
	0xf2
	0xfa
0xa1	0x80
0xa2	0x80
0xa3	0x80
0xa4	0x80
0xa5	0x80
0xa6	0x80
0xa7	0x80
0xa8	0x80
0xa9	0x80
0xaa	0x80
0xab	0x80
0xac	0x80
0xad	0x80
0xae	0x80
0xaf	0x80
0xb4	0x80
0xb5	0x80
0xb6	0x80
0xb7	0x80
0xb8	0x80
0xb9	0x40
	0x80
0xba	0x00
	0x40
	0x70
	0x7f
	0x80
	0x81
	0x82
	0x83
	0x84
	0x86
	0x87
	0x88
	0x8c
	0x8f
	0x90
	0x98
	0x9e
	0xa0
	0xb0
	0xbc
	0xc0
	0xc0
	0xe0
	0xf8
0xbb	0x00
	0x60
	0x80
0xbc	0x80
0xbd	0x00
	0x80
0xbe	0x80
0xbf	0x80
0xc4	0x80
0xc5	0x80
0xc6	0x80
0xc7	0x80
0xc8	0x80
0xc9	0x80
0xca	0x80
0xcb	0x80
0xcc	0x80
0xcd	0x80
0xce	0x80
0xcf	0x80
0xd4	0x80
0xda	0x80
0xe0	0xd4
	0xe0
0xfa	0x80
0xfa	0x80
0xff	0x7f

Table 5.9: Extracted instruction set

Comparison with brute-force approach In this experiment, we found 522 distinct combinations of class, instruction, and parameters in about 30 minutes of fuzzing. Both class and instruction bytes ranged from `0x00` up to and including `0xff`. In order to compare performance with a brute-force approach, we assumed the brute-force approach has access to the same information, and is able to make decisions based on the returned status words.

The brute-force approach would have to generate 256 APDUs to determine all possible class bytes. It would have found 55 distinct classes, and then have to generate, 256 APDUs per class to find all possible instructions. This gives $56 \cdot 256 = 14\,336$ APDUs in total to generate.

We saw in the beginning of this section (Section 5.5.2) that we could send about 17 APDUs per second to the smartcard on average. Sending 14 336 APDUs takes 843 seconds, or just over 14 minutes. This means our fuzzing approach using status words as side-channel information is a *less* viable one in terms of time spent in comparison with a brute-force approach that has access to, and can act upon, the same information.

5.5.3 Discussion

Error messages Table 5.7 shows the distribution of status words returned from the fuzzing session. We briefly elaborate on the interpretation of these three different messages:

1. No EF selected (`0x9400`)
2. No error (`0x9000`)
3. Proactive SIM (2x: `0x9138`, `0x9111`)

The first message refers to an *Elementary File*[19]. These files contain information like the phone number, unique identifier of the SIM card, etc. This means we hit a command that wants to perform an action on an EF.

The second message is interesting, because this means a command was completed successfully. The next step would be to look up the exact instructions with the 63 returned status words, and relate them to the GSM specification, but this is beyond the scope of this experiment.

The third message is part of a 'proactive SIM' command. These commands can be sent by the SIM card to the phone. The last byte indicates the length of the response data. More information can be found in the GSM 11.14[18] specification.

Logging input and output Our communication interface program (Section 5) logs APDUs sent, and status words received. It also reports if an error occurred after an APDU was sent. However, each of these log lines were separate. Also, we logged generic statistics every 10 seconds, interspersing regular log lines with statistics. We wrote a small log parser that searches for a sign of sending an APDU, and then looks for the corresponding result in the next lines. Additionally, this log parser splits the APDU into class, instruction, and parameter bytes, for easy processing afterwards. It would have been better to combine the generated APDU, and either error message or status words into one log message.

5.5.4 Conclusion

In this experiment, we wanted to find the instruction set of a smartcard, using status words as pseudo side-channel. Status words indicate the executed code path, and if an instruction is implemented. We considered an instruction as 'working' when the smartcard did not indicate the class, instruction, or parameters were incorrect. We used a non-personalized SIM card as fuzzing target, vanilla AFL, and our communication interface program as described in Section 5 (Supporting software).

We found out that this approach works in finding a working instruction set of a smartcard.

Our fuzzing approach is a *less* time-efficient one in comparison with a brute-force approach that has access to, and is able to act upon, the returned status words. Whereas our fuzzing approach took about 30 minutes to complete, the brute-force approach would have done the same in about 14 minutes.

5.6 Experiment 6: Status words, timing, `AFL`, `PasswordEq`

In this experiment, we fuzz a Java Card applet on a smartcard using a timing side-channel (Section 2.2.1) in addition to the pseudo side-channel status words (Section 2.3.3, see also Experiment 4 in Section 5.4). The status words report if the supplied password length was correct, and if the password as a whole was correct or incorrect. We use the `PasswordEq` applet (Section 4.1) that contains a password of fixed length, and two functions to check an input with the stored password. One function is timing-sensitive, the other executes in constant time. Ideally, `AFL` uses the timing information to zoom in on the timing-sensitive function, because this function leaks how many characters were correct through the timing side-channel. This information reduces the search space (worst case) from 256^{10} possibilities to $256 \cdot 10$, giving linear search time rather than exponential. We expect that the fuzzer finds the correct password length, and ideally also the correct password.

We supply timing information from the execution of the `PasswordEq` applet (Section 4.1) to `AFL`. The idea is that `AFL` interprets the duration of an execution as 'the fuzzing target visited a(nother) code path'. Given that `AFL` tries to find all code paths, this interpretation of timing information leads to `AFL` finding all possible execution times, including the one where it finds the correct password.

We verify that `AFL` uses the timing side-channel by comparing results of this experiment with those of Experiment 4 (Section 5.4). The setup is identical, but we add timing information in this experiment. We compare the distribution of status words, specifically those returned by the password checking instructions (timing-sensitive, and constant-time, see Section 4.1). We expect that, due to `AFL` using the timing information for mutating inputs, the timing-sensitive password checking instruction is called more often, relatively speaking, than the constant-time one.

5.6.1 Approach

We use the `PasswordEq` applet (Section 4.1) as fuzzing target for vanilla `AFL`. This applet contains a purpose-built timing side-channel. We capture status words (see Section 5.4) as well as the applet's execution time (explained further on) in `AFL`'s shared memory. We use separate memory regions to prevent overlap and possible interference between the two information sources.

We capture the applet's execution time, specifically the time between sending an `APDU` and receiving the status words with our communication interface program (Section 5). This duration is measured in nanoseconds, because we concluded from our previous experiment (Section 5.3) that millisecond precision lacks distinguishability between executions. The duration of an execution is encoded in AFL's shared memory by filling up memory as the duration increases.

We determined that a single execution of a password checking function takes about 11 ms. With nanosecond precision, and the amount of memory we could fill, we could only register up to 6 ms of execution time. Therefore, we simply lowered the precision to 10 ns (nanoseconds). This precision allows up to 60 ms of execution time to be registered in AFL's shared memory. This is sufficiently large to capture a single execution.

For this experiment, we use non-modified AFL, version 2.95b, our own communication interface program (Section 5) for AFL—smartcard communication, a JavaCOS A40 Java Card smartcard containing the `PasswordEq` applet (Section 4.1), and the OmniKey CardMan 5121 smartcard reader. The overall architecture can be found in Figure 5.1.

5.6.2 Results

We ran a fuzzing session of 1 hour and 40 minutes. In this session, AFL generated 90 122 APDUs. Due to filter restrictions, only 76 838 APDUs were sent to the smartcard, and we received 67 169 status words from the smartcard. This gives a global hit rate of 74.5%. On average, we sent just under 13 APDUs per second to the smartcard, and received just over 11 status words per second.

The distribution of status words returned from the `PasswordEq` applet can be found in Table 5.10.

Status message	**Status words**	**Count**	**Notes**
Class not supported	`0x6E00`	28 171	
Wrong P1/P2	`0x6A86`	17 363	
Wrong length	`0x6700`	15 667	
Instruction not supported	`0x6D00`	3 102	
Incorrect password	`0x66FF`	2 859	`INS_TRY_PASSWORD_UNSAFE` (`0x80`)
Incorrect password	`0x66FF`	7	`INS_TRY_PASSWORD_SAFE` (`0x82`)
Total		67 169	

Table 5.10: Returned status words by the `PasswordEq` applet, sorted on occurrence

5.6.3 Discussion

Comparison with Experiment 4 We first give a general comparison between this experiment and Experiment 4 (Section 5.4). Then, we verify that **AFL** uses the timing side-channel, by comparing the results, specifically the *Incorrect password* status words, in this experiment with the results from Experiment 4.

This experiment continued on the approach of Section 5.4, and added timing information as side-channel information. The fuzzing session in this experiment was shorter (1 hour, 40 minutes versus 2.5 hours), and **AFL** generated far less **APDU**s in this experiment (90 122 versus 2 101 613). This is also reflected in the average number of **APDU**s sent, and status words received in this experiment (13 and 11, respectively versus 129 and 92, respectively). We think the lower throughput (**APDU**s per second) is due to the additional timing information that **AFL** needs to process.

Now, we verify that **AFL** uses the timing side-channel by comparing the results from this experiment with Experiment 4 (Section 5.4). The only difference with Experiment 4 is the addition of the timing side-channel in this experiment. We specifically look at the status words returned by the two password checking instructions (`0x80`: 3 959 in Experiment 4 vs. 2 859 here, `0x82`: 165 in Experiment 4 vs. 7 here, see Table 5.6 in Section 5.4, and Table 5.10 here. Relatively speaking, in Experiment 4, 96% of all the *Incorrect password* status words were returned by the timing-sensitive password checking function (`0x80`). In this experiment, this figure is 99.8%, an increase of 3.8 percentage point. We expected that, by *adding* information from the timing side-channel to **AFL**, the timing-sensitive password checking function was called more often, relatively speaking, which indicates that **AFL** uses the timing side-channel to mutate inputs. Otherwise, we would have expected no change. Our results (96% vs. 99.8%) confirm that **AFL** uses the timing side-channel for mutating its inputs.

Password length and password correctness The `PasswordEq` applet leaks information about the password through status words and a timing side-channel. In Section 4.1, we saw that the applet returns the status words *Incorrect length* when the supplied length parameter is incorrect, and, if the length parameter is correct, returns the status words *Incorrect password* (or *Correct password*) depending on the correctness of the whole password. We see that **AFL** is able to guess the password length, because the applet returned *Incorrect password.* However, the applet did not return at least one status word indicating *Correct password.* This means that **AFL** did not find the correct password.

We think there are several factors that contributed to this result. First of all, the length of the stored password, and the possibilities per 'character'. In our case, we set the length at 10 bytes, giving each 'character' 256 possibilities: 256^{10}, or about $1.2 \cdot 10^{24}$ combinations. It would have been better to limit the total number of combinations, either by decreasing the total password length, or limiting the number of possibilities per 'character'. Secondly, although AFL found the correct length value, when looking into the actual APDUs sent, AFL did not use this exact length when guessing the password. Frequently, AFL generated an APDU which passes the 'correct length' check, but did not contain that amount (10) of password bytes. We expected that both a check on the length of the internal buffer as well as on the length parameter (`Lc`) would catch such invalid APDUs[12]. Contrary to what we expected, the reported length of the APDU buffer has no connection to the actual length of the APDU according to the Java Card documentation[2]. This first length check could have leaked information about the correct password length, both through timing as well as status words. This would have provided AFL with the information we expected it to get.

Better allocation of AFL's shared memory We could improve on the shared memory split, because in this experiment, we used just under half of the memory capacity for timing. Consequently, we had to make concessions on the timing precision. The other half is sparsely used, because the number of possible status words returned by the applet is six (Section 4.1.1). A better approach would be to map the possible status words to a distinct memory region, and use the rest of the memory for timing information.

5.6.4 Conclusion

In this experiment, we fuzzed a Java Card applet on a smartcard using a timing side-channel (Section 2.2.1) in addition to the pseudo side-channel status words (Section 2.3.3, see also Experiment 4 in Section 5.4). We used the `PasswordEq` applet (Section 4.1) that contains a password of fixed length, and two functions to check an input with the password. One function is timing-sensitive, the other executes in constant time. We expected that AFL finds the correct password length and, ideally, finds the correct password using the timing side-channel.

[12]See Code B.1, lines 91–99

Our results show that `AFL` found the correct password length. We also found that `AFL` uses the timing side-channel, but this extra information did not lead to `AFL` finding the correct password. We attribute this to the complexity of the stored password (too long, too many combinations per character), as well as a flaw in one of the password length checks. This flaw prevented `AFL` from knowing if the length of the *actual* password sent, rather than the reported length (`Lc`, see Section 5.6.3 for more details), was correct.

Chapter 6

Future work

In this chapter, we give ideas for future work.

Fully automated instruction set discovery using side-channels In Section 5.5, we created a setup that discovered, in a black-box approach, implemented instructions on a smartcard more efficiently than a brute-force approach. However, we used a *pseudo* side-channel: status words. Besides that, analyis needed to be done manually. Future work could include both using a real side-channel (e.g. timing or power usage), as well as creating an analysis tool (or simple processing pipeline) that automatically formats results. The former would improve on the ability to fuzz *other* (embedded) hardware than smartcards, whereas the latter makes it more easy to report results on a multitude of fuzzing targets.

Feeding a power side-channel to AFL In this thesis, we only considered a timing side-channel for integration with AFL. We briefly mentioned power side-channels in the preliminaries, and built a preliminary setup for a power tracing experiment. Future work could study the multi-dimensional information stream that a power tracer captures, and think of methods for integrating that into AFL.

White-box instrumentation of Java Card applets Similar to AFL's instrumentation of PC binaries, adding instrumentation to Java Card applets could benefit the fuzzing process on that platform. For example, on a PC, DifFuzz instruments the fuzzing target beforehand, in order to be able to count the number of executed branching instructions. The same approach could be done with a Java Card applet: instrument the source code, for example by injecting a branch counter near every branching instruction, recompile the applet, and install it on the smartcard. However, the instrumentation results need to be communicated on the same communication channel as 'regular' input and output of the smartcard. This would probably require a wrapper protocol implemented both on the fuzzer side, as well as on the applet, to accommodate both regular protocol, and transfer of results. This could be evaded if either the source or compiled form are available, and an emulator on PC is available that can be modified to implement the instrumentation part.

Chapter 7

Conclusion

The subject of this thesis was side-channel assisted fuzzing in embedded systems. We researched this subject, because analysing the internal behaviour of embedded systems is typically difficult. Software may not be available, either in source or in binary form, and attaching a debugger is typically not possible. In terms of analysing these embedded systems, we need to look at (pseudo) side-channels to gather information about the execution paths of the system.

We considered the following research questions:

1. What fuzzers are already using side-channel information?
2. How can we safely interface a Java Card smartcard with the fuzzer **AFL**?
3. How can we provide the fuzzer **AFL** with side-channel information?

For the first research question, we found that the differential fuzzer **DifFuzz** [24] uses the side-channel 'number of executed branching instructions' in Java programs for PC. In Section 5.1, we repeated a subset of their experiments to verify their approach. In our evaluation, we found similar results, despite having a different hardware and software setup.

For the second research question, we implemented a communication interface program that communicates with **AFL** on the one hand, and with the Java Card smartcard on the other hand (Section 5). The interface program provides a filtering mechanism, such that we can prevent certain inputs generated by **AFL** from reaching the smartcard. For example, too long inputs can be discarded, or inputs that would modify internal state.

For the third research question, we used the setup from research question two, and created a custom Java Card applet (Section 4.1) that contained a password checker. This password checker was implemented in both a timing-sensitive and constant-time way. We provided AFL with the *pseudo* side-channel status words, in order to find a stored password (Section 5.4). We found that AFL automatically detected both the correct instructions to test passwords and the correct password length, but did not find the stored password. We used status words again in AFL to find an instruction set of a SIM card (Section 5.5). With half an hour of fuzzing time, AFL found a complete instruction set, whereas a similar brute-force approach under the same conditions would take just under 14 minutes. Finally, we added timing information, in addition to status words, of the Java Card applet to AFL (Section 5.6). We found that AFL used the timing side-channel and was able to find the correct password length, but not the correct password.

In conclusion, we let the fuzzer AFL safely fuzz a smartcard, aided by the smartcard's side-channel information: the pseudo side-channel status words, and a timing side-channel. In our experiments with a custom password checking Java Card applet, we found that status words as side-channel information for AFL lets AFL find the length of the stored password, but not the stored password itself. Adding timing information on top of that did not result in finding the stored password. In our experiment with a SIM card, we found that status words alone are suitable to find the instruction set of the smartcard, but the approach is twice as slow as a brute-force approach under the same conditions.

Chapter 8

Acknowledgements

I would like to thank the following people who supported me throughout, or at some intermediate stage of, this thesis project. Thanks to Niels Samwel, for reviewing early work on this thesis. Thanks to Łukasz Chmielewski, for helping with the setup of a power trace analysis, which unfortunately did not make it into the final version of the thesis. Thanks to Daan Sprenkels for reviewing the final draft of this thesis, as well as being available for bouldering sessions, lunches in the park, *borrels* in the student canteens (when they were still open), side-projects, and many things more.

Also thanks to the Digital Security department of the Institute for Computing and Information Sciences of the Radboud University, for providing the fuzzing hardware and materials used in Chapter 5: Experiments. Thanks to Erik Poll and Lejla Batina for their work as supervisor and second reader.

Finally, I would like to thank Judith van Stegeren for supporting me on and with all aspects throughout this thesis project.

Bibliography

[1] american fuzzy lop. `http://lcamtuf.coredump.cx/afl/`. Fuzzer used in this thesis.

[2] APDU (Java Card API and Subsets). `http://www.unsads.com/specs/javacard/2.2.1/javacard/framework/APDU.html#getBuffer()`. Java Card API documentation on getting the buffer contents of an APDU.

[3] Clock (Java Platform 8 SE). `https://docs.oracle.com/javase/8/docs/api/java/time/Clock.html#systemUTC--`. Java API documentation on the available system clocks.

[4] *EMV Integrated Circuit Card Specifications for Payment Systems, Book 1–4.* via `https://www.emvco.com/`.

[5] GlobalPlatformPro. `https://javacard.pro/globalplatform/`. GlobalPlatform-Pro provides a simple, commandline interface program for interacting with smart-cards.

[6] Instant (Java Platform 8 SE). `https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html#now--`. Java API documentation on querying the system clock for the current time.

[7] isstac/diffuzz. `https://github.com/isstac/diffuzz`. Source code repository of the differential fuzzer called DifFuzz, specifically aimed at fuzzing Java programs.

[8] isstac/diffuzz at v1.0.0-citable. `https://github.com/isstac/diffuzz/tree/v1.0.0-citable`. Differential fuzzer called DifFuzz. This links to a specific snapshot of the source code repository used in this thesis.

[9] Java Archive Downloads - Java ME. https://www.oracle.com/java/technologies/java-archive-downloads-javame-downloads.html#java_card_kit-2.2.1-oth-JPR. Java Card 2.2.1 software development kit used in this thesis.

[10] JavaCOS A40 dual interface Java card - 64K. https://www.smartcardfocus.com/shop/ilp/id~711/javacos-a40-dual-interface-java-card-64k/p/index.shtml. Java Card smartcard used for experiments in this thesis.

[11] List of supported JavaCard algorithms. https://www.fi.muni.cz/~xsvenda/jcalgtest/table.html. This website lists the capabilities regarding cryptographic algorithms supported by various types of Java Card smartcards.

[12] PCSClite project. https://pcsclite.apdu.fr/. PCSClite is used in Linux to communicate with smartcard terminals, and smartcards.

[13] sf1/go-card: PC/SC client written in Go. https://github.com/sf1/go-card/tree/master. This library provides an API for interacting with a smartcard using the PCSC-lite daemon.

[14] System (Java Platform 8 SE). https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#nanoTime--. Java API documentation on querying the current timestamp in nanoseconds.

[15] Technical ”whitepaper” for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt. Technical description of the internals of the fuzzer american fuzzy lop (afl).

[16] F. Aarts, J. De Ruiter, and E. Poll. Formal models of bank cards for free. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 461–468, March 2013.

[17] Barton Milller. Foreword for Fuzz Testing Book. April 2008.

[18] ETSI TC-SMG. Digital cellular telecommunications system (Phase 2+); Specification of the SIM Application Toolkit for the Subscriber Identity Module - Mobile Equipment (SIM - ME) interface (GSM 11.14). Standard, European Telecommunications Standards Institute, 1996.

[19] ETSI TC-SMG. Digital cellular telecommunications system (Phase 2+); Specification of the Subscriber Identity Module - Mobile Equipment (SIM - ME) interface (GSM 11.11). Standard, European Telecommunications Standards Institute, 1996.

[20] ISO Central Secretary. Identification cards — Integrated circuit cards — Part 4: Organization, security and commands for interchange. Standard ISO/IEC 7816-4:2013, International Organization for Standardization, Geneva, CH, 2013.

[21] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, 2018.

[22] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, Sep. 2018.

[23] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[24] Shirin Nilizadeh, Yannic Noller, and Corina S. Păsăreanu. Diffuzz: Differential fuzzing for side-channel analysis. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 176–187, Piscataway, NJ, USA, 2019. IEEE Press.

[25] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security Privacy*, 3(2):58–62, March 2005.

[26] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*, February 2017.

[27] Henning Richter, Wojciech Mostowski, and Erik Poll. Fingerprinting passports. In *NLUUG spring conference on security*, volume 1, 2008.

[28] Dennis Vermoen, Marc Witteman, and Georgi N. Gaydadjiev. Reverse engineering java card applets using power analysis. In Damien Sauveron, Konstantinos Markantonakis, Angelos Bilas, and Jean-Jacques Quisquater, editors, *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems*, pages 138–149, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

Appendix A

Smartcard communication

In this chapter, we give an overview on how to setup communication with a smartcard (Section A.1), an introduction into the GlobalPlatformPro toolset for interacting with the smartcard (Section A.2), and how to upload Java Card applets (Section A.3).

We use the *OmniKey CardMan 5121* smartcard reader (terminal), which connects through USB to a PC. Our smartcard is the JavaCOS A40[10], a Java Card with 64 KB storage and 1.6 KB of memory. Although both the smartcard and smartcard terminal support contactless communication, we only use the contact-based communication method.

The reader of this chapter should be familiar with the Linux command line. We assume the reader has root access to the machine for installing packages, starting system processes, and performing tasks that require such elevated privileges.

A.1 Communication setup

The machine we use[1] runs Linux. Communication with a smartcard reader on this platform is managed through the `pcsclite`[12] suite. It comprises a background process (daemon) called `pcscd` that takes care of communication with the hardware (i.e. the reader and the smartcard), and a set of tools to communicate with `pcscd`.

In our Linux distribution (*Arch Linux*), we install the packages `pcsclite` and `pcsc-tools`. We make sure the daemon runs (as root: `# systemctl start pcscd`, or manually

[1]Lenovo x230, Intel i5-3320M @ 2.60 GHz, 16 GB RAM

`# pcscd -f`). Next, we run, as root, `# pcsc_scan`, such that we can observe the communication between the smartcard and the reader.

We now assume the OmniKey CardMan 5121 is connected as the sole smartcard reader and `pcsc_scan` runs (as root). When we insert the JavaCOS A40 card, we see the following output:

```
 Reader 1: OMNIKEY AG CardMan 5121 01 00
  Event number: 7
  Card state: Card inserted,
  ATR: 3B 9F 95 81 31 FE 9F 00 66 46 53 05 10 00 FF 71 DF 00 00 00 00 00 EC
```

Code A.1: Output of `pcsc_scan` when inserting the JavaCOS A40 smartcard

`pcsc_scan` also tries to interpret the `ATR` ("Answer To Reset") bytes which are shown in hexadecimal form in the last line of Code A.1). *Answer To Reset* contains the communication settings of the card. The smartcard reader interprets this in order to communicate correctly with the card.

At the end of the output, it says:

```
Possibly identified card (using /usr/share/pcsc/smartcard_list.txt):
3B 9F 95 81 31 FE 9F 00 66 46 53 05 10 00 FF 71 DF 00 00 00 00 00 EC
  JavaCOS A22 dual interface Java card - 150K (JavaCard)
  http://www.smartcardfocus.us/shop/ilp/id~709/javacos-a22-dual-interface-java-card-150k/p/index.
     shtml
```

Code A.2: ATR (Answer-to-Reset) output of the JavaCOS A40 smartcard

We expected it to be identified as the JavaCOS A40, because it says so on the added label (the card itself has no identifying features). The exact `ATR` can also be found on the *JCAlgTest* web page[11], where it is identified as the *Feitian A40*. This could mean that the same `ATR` describes multiple cards. We briefly show this in the following example, with another smartcard:

```
 Reader 0: OMNIKEY AG CardMan 5121 00 00
  Event number: 1
  Card state: Card inserted,
  ATR: 3B 67 00 00 29 20 00 6F 78 90 00
[..]
Possibly identified card (using /usr/share/pcsc/smartcard_list.txt):
3B 67 00 00 29 20 00 6F 78 90 00
  ING (previously Postbank Chippas) (chipknip) Netherlands
  Rabobank bankcard (dutch)
  ASN Bank debit card
  SNS Bank debit card
```

```
ABN-AMRO Maestro
```

Code A.3: ATR (Answer-to-Reset) output of an ING banking card

The card inserted was an ING banking card. This example shows that the same `ATR` is used in cards from different banks.

A.2 Tools: GlobalPlatformPro

We use the *GlobalPlatformPro* toolset, available at [5], for interacting with the smartcard. Throughout this section, we use the alias[2] `gp-card` to prevent confusion with PARI/GP, the computer algebra system, that has `gp` as command name.

After installing the GlobalPlatformPro tools, we can acquire information about the smartcard with `gp-card --info`.

Note: *any* PC/SC compatible smartcard or device connected with the PC can answer to `gp-card --info`. For example, a hardware security token like the *Yubico YubiKey* communicates through PC/SC. Be sure to disconnect other PC/SC-capable devices before continuing, or explicitly select the reader with `gp-card --reader name-of-the-reader`.

Output:

```
$ gp-card --info
GlobalPlatformPro 19.06.16-4-g1f6b677
Running on Linux 5.2.5-arch1-1-ARCH amd64, Java 11.0.5 by Oracle Corporation
Reader: OMNIKEY AG CardMan 5121 00 00
ATR: 3B9F958131FE9F00664653051000FF71DF0000000000EC
More information about your card:
    http://smartcard-atr.appspot.com/parse?ATR=3B9F958131FE9F00664653051000FF71DF0000000000EC

CPLC: ICFabricator=4090
      ICType=7794
      OperatingSystemID=86AA
      OperatingSystemReleaseDate=7311 (2017-11-07)
      OperatingSystemReleaseLevel=0187
      ICFabricationDate=8037 (2018-02-06)
      ICSerialNumber=1601B02A
      ICBatchIdentifier=0547
      ICModuleFabricator=4090
      ICModulePackagingDate=8037 (2018-02-06)
      ICCManufacturer=86AA
      ICEmbeddingDate=8037 (2018-02-06)
      ICPrePersonalizer=86AA
      ICPrePersonalizationEquipmentDate=8037 (2018-02-06)
```

[2]For the Linux shell `bash`: `alias gp-card='java -jar /usr/share/java/globalplatformpro/gp.jar'`

```
      ICPrePersonalizationEquipmentID=00000000
      ICPersonalizer=0000
      ICPersonalizationDate=0000 (2010-01-01)
      ICPersonalizationEquipmentID=00000000

Card Data:
Tag 6: 1.2.840.114283.1
-> Global Platform card
Tag 60: 1.2.840.114283.2.2.1.1
-> GP Version: 2.1.1
Tag 63: 1.2.840.114283.3
Tag 64: 1.2.840.114283.4.2.85
-> GP SCP02 i=55
Tag 65: 1.3.656.840.100.2.1.3
Tag 66: 1.3.6.1.4.1.42.2.110.1.2
-> JavaCard v2
Card Capabilities:
Version: 255 (0xFF) ID:   1 (0x01) type: DES3 length:  16
Version: 255 (0xFF) ID:   2 (0x02) type: DES3 length:  16
Version: 255 (0xFF) ID:   3 (0x03) type: DES3 length:  16
Key version suggests factory keys
```

Code A.4: Output of `gp-card --info` for the JavaCOS A40 smartcard

We can also acquire a list of applets installed on the smartcard using `gp-card --list`:

```
$ gp-card --list
Warning: no keys given, using default test key 404142434445464748494A4B4C4D4E4F
ISD: A000000003000000 (OP_READY)
     Privs:   SecurityDomain, CardLock, CardTerminate, CardReset, CVMManagement

APP: 4141414141FF (SELECTABLE)
     Privs:

PKG: 4141414141 (LOADED)
     Version: 1.0
     Applet:  4141414141FF
```

Code A.5: Output of `gp-card --list` of the JavaCOS A40 smartcard

A.3 Applet installation

The output above, Code A.5 shows an applet that has already been installed, identified by the *applet AID* `4141414141`. In order to install an applet on the JavaCard, we need a *Converted Applet*, or `CAP` file. This conversion is done by a tool from the *Java Card Development Kit*. We use version 2.2.1 which can be acquired from [9]. Unpack the archive in a directory. This directory should also contain the `Makefile` and a couple of other directories, see further on.

In Code A.6, we provide a `Makefile` that makes sure our JavaCard applet can be built from source and installed on the smartcard. The `Makefile` has been adapted from the one provided through the *Hardware Security* course at the Radboud University.

The `Makefile` uses the GlobalPlatformPro toolset to install the `CAP` file on the smartcard. `gp-card` is called through an internal reference (see line 11 in Code A.6); adapt it where necessary, or make sure `gp.jar` is in the subdirectory `gp/` relative from where the `Makefile` is located. The name of the applet (`APPLET_NAME`, line 1) is leading for the directory structure, and where to place the applet's source code. To create that structure, run `$ APPLET_NAME=yourappletname mkdir -p ${APPLET_NAME}/{bin,src}/{applet,javacard}`, and put your applet's source code in `${APPLET_NAME}/src/applet/${APPLET_NAME}.java`.

In order to build and install the applet, run `$ make applet` in the directory where the `Makefile` is located. To remove the built files, run `$ make clean`.

```
1  APPLET_NAME=PasswordEqApplet
2  APPLET_AID="0x41:0x41:0x41:0x41:0x41"
3  APPLET_SFX="0xFF"
4  APPLET_VERSION="1.0"
5
6  JC_HOME=java_card_kit-2_2_1
7
8  JC_PATH=${JC_HOME}/lib/apdutool.jar:${JC_HOME}/lib/apduio.jar:${JC_HOME}/lib/converter.jar:${
       JC_HOME}/lib/jcwde.jar:${JC_HOME}/lib/scriptgen.jar:${JC_HOME}/lib/offcardverifier.jar:${
       JC_HOME}/lib/api.jar:${JC_HOME}/lib/installer.jar:${JC_HOME}/lib/capdump.jar:${JC_HOME}/
       samples/classes:${CLASSPATH}
9
10 CONVERTER=java -Djc.home=${JC_HOME} -classpath ${JC_PATH}:${APPLET_NAME}/bin com.sun.javacard.
       converter.Converter
11 GP=java -jar gp/gp.jar
12
13 all:
14
15 clean:
16   rm -rf ./${APPLET_NAME}/bin/*
17   ${GP} --delete ${APPLET_AID}
18
19 applet: ./${APPLET_NAME}/bin/${APPLET_NAME}.class ./${APPLET_NAME}/bin/javacard/applet.cap
20
21 ./${APPLET_NAME}/bin/javacard/applet.cap: ${APPLET_NAME}/bin/${APPLET_NAME}.class
22   # Convert to CAP
23   ${CONVERTER} -v -out CAP -exportpath ${JC_HOME}/api_export_files \
24       -classdir ./${APPLET_NAME}/bin -d ./${APPLET_NAME}/bin \
25       -applet "${APPLET_AID}:${APPLET_SFX}" applet.${APPLET_NAME} \
26       applet ${APPLET_AID} ${APPLET_VERSION}
27
28   # Uninstall old applet
29   ${GP} --uninstall ./${APPLET_NAME}/bin/applet/javacard/applet.cap
```

```

  # Install new applet
  ${GP} --install ./${APPLET_NAME}/bin/applet/javacard/applet.cap

./${APPLET_NAME}/bin/${APPLET_NAME}.class: ${APPLET_NAME}/src/applet/${APPLET_NAME}.java
  # Compile the applet
  javac -source 1.3 -target 1.1 -d ${APPLET_NAME}/bin -cp ${JC_PATH} ${APPLET_NAME}/src/applet/${
     APPLET_NAME}.java
```
Code A.6: `Makefile` for compiling and uploading Java Card programs

Appendix B

Code listings

This appendix contains the most important code snippets used in this thesis. You can find the source code of the `PasswordEq` applet (Section 4.1) in Section B.1. The helper scripts, and fuzzing target used in Section 5.3 can be found in Section B.2. The source code of the communication interface program (Section 5) can be found in Section B.3.

The full source code is available from `https://ghcm.nl/` at the end of November 2020.

B.1 Java Card applet `PasswordEq`

In Code B.1, we find the source code of the Java Card applet, called `PasswordEq`, that checks a supplied password with the stored, static password. Depending on the instruction given (`0x80`, `0x82`), the applet uses an *unsafe* or *safe* method, respectively, for checking that supplied password. The unsafe method has a timing side-channel on the number of incorrect bytes in the supplied password, whereas the safe method does not. The applet is used in Section 5.2 for demonstration purposes, and as fuzzing target in Sections 5.3, 5.4, and 5.6.

```java
package applet;

import javacard.framework.APDU;
import javacard.framework.Applet;
import javacard.framework.ISO7816;

import javacard.framework.APDUException;
import javacard.framework.ISOException;
import javacard.framework.JCSystem;
import javacard.framework.SystemException;
```

```java

/**
 * PasswordEqApplet is a JavaCard applet that implements a static password checker in two variants.
 *
 * The caller should use class byte CLA=0xBA and give either of two instructions:
 * - the unsafe variant (INS=0x80)
 * - the safe variant (INS=0x82)
 * Both use a static command data length of 10 (Lc=0x0A)
 *
 * The applet returns in SW1-SW2 whether the password was correct or incorrect using 0x90FF and
 * 0x66FF, respectively. If an error occurred, a standard ISO7816-4 error status is returned.
 *
 * @author Gerdriaan Mulder <gmulder@science.ru.nl>
 */
public class PasswordEqApplet extends Applet implements ISO7816 {
    public static final byte INS_TRY_PASSWORD_UNSAFE = (byte)0x80;
    public static final byte INS_TRY_PASSWORD_SAFE   = (byte)0x82;

    public static final short RET_PASSWORD_INCORRECT = (short)0x66FF;
    public static final short RET_PASSWORD_CORRECT   = (short)0x90FF;

    private static final short PASSWORD_LENGTH = (short) 10;
    private byte[] cmdPassword;
    private byte[] savedPassword = {(byte)0x49, (byte)0x48, (byte)0x47, (byte)0x46,
                                    (byte)0x45, (byte)0x44, (byte)0x43, (byte)0x42,
                                    (byte)0x41,  (byte)0x40};

    /**
     * Constructor for the applet.
     */
    public PasswordEqApplet() {
        cmdPassword = JCSystem.makeTransientByteArray(PASSWORD_LENGTH, JCSystem.CLEAR_ON_RESET);
        register();
    }

    /**
     * install makes sure the applet is installed on the smartcard
     */
    public static void install(byte[] buffer, short offset, byte length)
            throws SystemException {
        new PasswordEqApplet();
    }

    /**
     * select performs operations when selecting the applet on the smartcard
     */
    public boolean select() {
        return true;
    }

    /**
     * deselect performs operations when deselecting the applet on the smartcard
     */
    public void deselect() {
    }

    /**
     * process receives a command APDU and performs operations based on the given instructions.
     *
     */
    public void process(APDU apdu) throws ISOException, APDUException {
        byte[] buffer = apdu.getBuffer();
        byte cla = buffer[ISO7816.OFFSET_CLA];
        byte ins = buffer[ISO7816.OFFSET_INS];
        byte p1  = buffer[ISO7816.OFFSET_P1];
        byte p2  = buffer[ISO7816.OFFSET_P2];
        short lc  = (short)0;

        // Ignore the "SELECT FILE" APDU (0x00A4)
        if (selectingApplet()) {
            return;
        }

        if(cla != (byte)0xBA) {
            ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
        }

        if(p1 != (byte)0x00 || p2 != (byte)0x00) {
            ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);
        }
        if(buffer.length < PASSWORD_LENGTH+5) {
            ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
        }

        lc = buffer[ISO7816.OFFSET_LC];
        if(lc != PASSWORD_LENGTH) {
            // We expect our predefined password length
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }

        // Fill cmdPassword buffer
        for(short i = 0; i < PASSWORD_LENGTH; i++) {
            cmdPassword[i] = buffer[(short)(ISO7816.OFFSET_CDATA+i)];
        }
        short passwordEq = 0;

        switch (ins) {
        case INS_TRY_PASSWORD_UNSAFE:
            // Compare byte-wise the password and return early when a character did not match.
            for(short i = 0; i < PASSWORD_LENGTH; i++) {
                if(cmdPassword[i] != savedPassword[i]) {
                    ISOException.throwIt(RET_PASSWORD_INCORRECT);
                }
            }
            break;
        case INS_TRY_PASSWORD_SAFE:
            // Add to passwordEq the stored password XOR the provided password. All characters are
    compared
            for(short i = 0; i < PASSWORD_LENGTH; i++) {
                //passwordEq += (short)(cmdPassword[i] ^ savedPassword[i]);
                passwordEq += (short)(cmdPassword[i] & 0x00FF) ^ (short)(savedPassword[i] & 0x00FF)
    ;
            }
            if (passwordEq != 0) {
                ISOException.throwIt(RET_PASSWORD_INCORRECT);
            }
            break;
        default:
            ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
        }

        ISOException.throwIt(RET_PASSWORD_CORRECT);
    }
}
```

Code B.1: Safe/unsafe password checker

B.2 Supporting scripts, and fuzzing target for Section 5.3

The following two code listings are related to Section 5.3: the experiment with DifFuzz and the `PasswordEq` applet. In Code B.2, you can find the shell script that compiles and instruments the fuzzing targets for DifFuzz. Code B.3 contains the overall fuzzing execution script. They should be executed in order, i.e. first the preparation script, then the execution script. Both scripts have been adapted from `evaluation/prepare.sh` and `evaluation/run_evaluation.sh` out of [8].

```
## prepare_javacard.sh
#####################################
# chmod +x prepare_javacard.sh
# ./prepare_javacard.sh
#

trap "exit" INT

# Prepare javacard_passwordEq_safe.
echo "Prepare javacard_passwordEq_safe.."
cd ./javacard_passwordEq_safe/
rm -rf bin
mkdir bin
cd src
javac -cp .:../../../tool/instrumentor/build/libs/kelinci.jar:../lib/* Driver_PCSC.java -d ../bin
cd ..
rm -rf bin-instr
```

```
java -cp ../../tool/instrumentor/build/libs/kelinci.jar:./lib/* edu.cmu.sv.kelinci.instrumentor.
     Instrumentor -mode LABELS -i ./bin/ -o ./bin-instr -skipmain
cd ..
echo ""

# Prepare javacard_passwordEq_unsafe.
echo "Prepare javacard_passwordEq_unsafe.."
cd ./javacard_passwordEq_unsafe/
rm -rf bin
mkdir bin
cd src
javac -cp .:../../../tool/instrumentor/build/libs/kelinci.jar:../lib/* Driver_PCSC.java -d ../bin
cd ..
rm -rf bin-instr
java -cp ../../tool/instrumentor/build/libs/kelinci.jar:./lib/* edu.cmu.sv.kelinci.instrumentor.
     Instrumentor -mode LABELS -i ./bin/ -o ./bin-instr -skipmain
cd ..
echo ""

echo "Done."
```

Code B.2: Shell script for preparing driver and fuzzing target for use in DifFuzz

```
## run_javacard_evaluation.sh
# CAUTION
# Run this script within its folder. Otherwise the paths might be wrong!
#####################################
# chmod +x run_javacard_evaluation.sh
# ./run_javacard_evaluation.sh
#

trap "graceful_exit" INT

server_pid="-1"
afl_pid="-1"

confirm() {
  # Ask user.
  read -p "Continue with evaluation? " -n 1 -r
  echo
  if [[ ! $REPLY =~ ^[Yy]$ ]]
  then
    echo "ABORT."
    exit 1
  fi
}

graceful_exit() {
  echo "Graceful exit"
  if [[ "$server_pid" != "-1" ]]; then
    echo "Killing server: $server_pid"
    kill $server_pid
  fi
  if [[ "$afl_pid" != "-1" ]]; then
    echo "Killing afl: $afl_pid"
    kill $afl_pid
  fi
  exit 0
}

####################

number_of_runs=5
time_bound=1800 #30min
step_size_eval=30

declare -a subjects=(
"javacard_passwordEq_safe" # JavaCard interface
"javacard_passwordEq_unsafe" # JavaCard interface
)

declare -a classpaths=(
"./bin-instr/" # "javacard_passwordEq_safe"
"./bin-instr/" # "javacard_passwordEq_unsafe"
)

declare -a drivers=(
"Driver_PCSC" # "javacard_passwordEq_safe"
"Driver_PCSC" # "javacard_passwordEq_unsafe"
)


# Check array sizes
if [[ ${#subjects[@]} != ${#classpaths[@]} ]]
then
echo "[Error in script] the array sizes of subjects and classpaths do not match!. Abort!"
exit 1
fi
if [[ ${#subjects[@]} != ${#drivers[@]} ]]
then
echo "[Error in script] the array sizes of subjects and drivers do not match!. Abort!"
exit 1
fi

NUMsubj=${#subjects[@]}
ETAsec=$((${NUMsubj} *${number_of_runs}*${time_bound}))
echo -n "I've seen ${NUMsubj} subjects. Estimate time to completion: ${NUMsubj} * ${number_of_runs}
     * ${time_bound} = ${ETAsec} seconds "
printf "(%.2f hours)\n" $((${ETAsec}/3600))

confirm

echo "Running evaluation as requested..."

subject_counter=0
total_number_subjects=${#subjects[@]}
```

```
total_number_experiments=$(( $total_number_subjects * $number_of_runs))

for (( i=0; i<=$(( $total_number_subjects - 1 )); i++ ))
do
  cd ./${subjects[i]}/

  for j in `seq 1 $number_of_runs`
  do

    run_counter=$(( $run_counter + 1 ))
    echo "[$run_counter/$total_number_experiments] Run analysis for ${subjects[i]}, round $j .."

    # Start Kelinci server
    nohup java -cp ${classpaths[i]} edu.cmu.sv.kelinci.Kelinci ${drivers[i]} @@ > ./server-log-$j.
     txt &
    server_pid=$!

    # Start modified AFL
    AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1 AFL_SKIP_CPUFREQ=1 nohup ../../tool/afl-2.51b-wca/afl-
     fuzz -i in_dir -o fuzzer-out-$j -c userdefined -S afl -t 999999999 ../../tool/fuzzerside/
     interface @@ > ./afl-log-$j.txt &
    afl_pid=$!

    # Wait for timebound
    sleep $time_bound

    # Stop AFL and Kelinci server
    kill $afl_pid
    kill $server_pid

    # Wait a little bit to make sure that processes are killed
    sleep 10

  done

  cd ..

  # Evaluate run
  python3 evaluate_cost.py ${subjects[i]}/fuzzer-out- $number_of_runs $time_bound $step_size_eval

done
```

Code B.3: Shell script for running the fuzzing process

B.2.1 Fuzzing target

The next code listing is the *fuzzing target* for DifFuzz. This part is responsible for connecting to a smartcard inserted into a smartcard reader, feeding the fuzzing input to the fuzzing target, and recording the elapsed time between sending the Command APDU, and receiving the Response APDU in DifFuzz.

```
import java.lang.Exception;
import java.lang.String;
import java.time.Instant;
import java.time.Duration;
import java.util.List;

import javax.smartcardio.Card;
import javax.smartcardio.CardChannel;
import javax.smartcardio.CardException;
import javax.smartcardio.CardTerminal;
import javax.smartcardio.CardTerminals;
import javax.smartcardio.CommandAPDU;
import javax.smartcardio.ResponseAPDU;
import javax.smartcardio.TerminalFactory;

class PasswordEq_PCSC {
    public static final byte INS_TRY_PASSWORD_UNSAFE = (byte)0x80;
    public static final byte INS_TRY_PASSWORD_SAFE = (byte)0x82;
    public static final short RET_PASSWORD_CORRECT = (short)0x90FF;
    public static final short RET_PASSWORD_INCORRECT = (short)0x66FF;
    public static long passwordEq_cost = 0;

    public static final byte[] PASSWORDEQ_AID = {
        (byte)0x41, (byte)0x41, (byte)0x41, (byte)0x41, (byte)0x41, (byte)0xFF
    };
    // ISO7816 select command APDU, class 0x00, instruction 0xA4, P1 0x04, P2 0x00, and applet ID
    public static CommandAPDU APDU_SELECT = new CommandAPDU(0x00, 0xA4, 0x04, 0x00, PASSWORDEQ_AID)
     ;

    // checkPassword_safe returns whether the password was correct. It sets the passwordEq_cost
    // such that the fuzzer knows the cost of the operation. Our cost is defined as the "time spent
    // checking the password, in nanoseconds"
    public static boolean checkPassword_safe(byte[] guess, byte[] pw) {
        return checkPassword(true, guess, pw);
    }

    // checkPassword_unsafe returns whether the password was correct. It sets the passwordEq_cost
    // such that the fuzzer knows the cost of the operation. Our cost is defined as the "time spent
    // checking the password, in nanoseconds"
    public static boolean checkPassword_unsafe(byte[] guess, byte[] pw) {
        return checkPassword(false, guess, pw);
    }

    private static boolean checkPassword(boolean safe, byte[] guess, byte[] pw) {
        // Setup communication with the card reader, only consider terminals with an inserted card
        short appletReturn;
        try {
            TerminalFactory termfactory = TerminalFactory.getDefault();
            CardTerminals cardTerms = termfactory.terminals();
            List<CardTerminal> terminals = cardTerms.list(CardTerminals.State.CARD_PRESENT);
            try {
                if (terminals.isEmpty()) {
```

```java
                throw new RuntimeException("Cannot setup communication with reader: " + cardTerms);
            }
        } catch (Exception e) {
            System.err.println("exception occurred: " + e);
            return false;
        }

        // Select the first terminal and connect with the card
        CardTerminal t = terminals.get(0);
        Card c = t.connect("*");
        CardChannel chan = c.getBasicChannel();

        // Select the applet
        ResponseAPDU reply = chan.transmit(APDU_SELECT);
        try {
            if ((short)reply.getSW() != (short)0x9000) {
                throw new Exception("Cannot SELECT PasswordEq applet: " + reply);
            }
        } catch (Exception e) {
            System.err.println("exception occurred: " + e);
            return false;
        }

        byte instruction = INS_TRY_PASSWORD_UNSAFE;
        if (safe) {
            instruction = INS_TRY_PASSWORD_SAFE;
        }

        // Construct the APDU for trying a password with the supplied byte array, using the supplied instruction
        CommandAPDU tryPasswordAPDU = new CommandAPDU(0xBA, instruction, 0x00, 0x00, pw);

        // Query the applet and save the output, record the time spent
        long now = System.nanoTime();
        reply = chan.transmit(tryPasswordAPDU);
        long afterQuery = System.nanoTime();

        try {
            appletReturn = (short)reply.getSW();

            // avoid && short-circuit
            // grep 'Unexpected return code' server-log-1.txt | sort | uniq -c
            if (!(appletReturn == RET_PASSWORD_CORRECT || appletReturn == RET_PASSWORD_INCORRECT)) {
                byte low = (byte)(appletReturn & 0xff);
                byte high = (byte)((appletReturn >> 8) & 0xff);
                throw new Exception("Unexpected return code: (" + String.format("%x", appletReturn) + ") 0x"+ String.format("%02x%02x", high, low));
            }
        } catch (Exception e) {
            System.err.println("exception occurred: " + e);
            return false;
            }

            // Set time spent as passwordEq_cost
            passwordEq_cost = Math.abs(afterQuery - now);

        } catch (CardException e) {
            System.err.println("Unhandled exception: " + e);
            return false;
        }

        return appletReturn == RET_PASSWORD_CORRECT;
    }

    // clearCost resets the passwordEq_cost variable
    public static void clearCost() {
        passwordEq_cost = 0;
    }

    public static long getCost() {
        return passwordEq_cost;
    }
}
```

Code B.4: Fuzzing target implementation for using a smartcard with DifFuzz

B.2.2 Sign extension issues from Section 5.3

Code B.5 shows the effect of improper handling of sign extension, as discussed in Section 5.3.3: **Logical, and sign extension bug**. See also the commented `passwordEq` variable in Code B.1, line 119, which caused this behaviour.

```
package main

import (
  "fmt"
)

func main() {
  var eq int16
  saved := []byte{0x49, 0x48, 0x47, 0x46, 0x45, 0x44, 0x43, 0x42, 0x41, 0x40}
  cmd := []byte{0x16, 0x45, 0x93, 0x35, 0x5b, 0x00, 0xf2, 0xff, 0xff, 0xff}
  for n, v := range saved {
    xor := int8(v ^ cmd[n])
    if xor <= 127 {
      eq += int16(xor)
      fmt.Printf("%#02x ^ %#02x = %+#02x (%4d, %4d): %08b^%08b=%08b\n", v, cmd[n], xor, xor, eq, v,
      cmd[n], xor)
    } else {
      eq += int16(-xor)
      fmt.Printf("%#02x ^ %#02x = %+#02x (%4d, %4d): %08b^%08b=%08b\n", v, cmd[n], xor, xor, eq, v,
      cmd[n], xor)
    }
  }
}

/* output:
0x49 ^ 0x16 = +0x5f (  95,   95): 01001001^00010110=01011111
0x48 ^ 0x45 = +0x0d (  13,  108): 01001000^01000101=00001101
0x47 ^ 0x93 = -0x2c ( -44,   64): 01000111^10010011=-0101100
0x46 ^ 0x35 = +0x73 ( 115,  179): 01000110^00110101=01110011
0x45 ^ 0x5b = +0x1e (  30,  209): 01000101^01011011=00011110
0x44 ^ 0x00 = +0x44 (  68,  277): 01000100^00000000=01000100
0x43 ^ 0xf2 = -0x4f ( -79,  198): 01000011^11110010=-1001111
0x42 ^ 0xff = -0x43 ( -67,  131): 01000010^11111111=-1000011
0x41 ^ 0xff = -0x42 ( -66,   65): 01000001^11111111=-1000010
0x40 ^ 0xff = -0x41 ( -65,    0): 01000000^11111111=-1000001
*/
```

Code B.5: Code demonstrating the sign extension bug in Section 5.3.3

B.3 Communication interface program

We wrote supporting software (see Section 5) to let **AFL** fuzz an applet on a smartcard. This section provides code snippets for the most important parts of that program.

The following two code listings take care of attaching to **AFL**'s shared memory (Code B.6), and setting up communication with **AFL**'s forkserver (Code B.7). These parts are used in Section 5.

```
package aflshm

import (
  "fmt"
  "io"
  "log"
  "os"
  "reflect"
  "strconv"
  "syscall"
  "unsafe"

  "golang.org/x/sys/unix"
)

const (
  SHM_SIZE = 65536
)

type SharedMem *[SHM_SIZE]byte

```

```go
func AttachSharedMemory(logger io.Writer) (SharedMem, error) {
  log.SetOutput(logger)

  pid, _, _ := unix.Syscall(unix.SYS_GETPID, 0, 0, 0)
  log.Printf("[%v] hello", pid)

  var shmptr uintptr
  var shared_mem SharedMem
  var shmHdr *reflect.SliceHeader
  var errno syscall.Errno

  shmname, in_afl := os.LookupEnv("__AFL_SHM_ID")
  if !in_afl {
    return shared_mem, fmt.Errorf("not running in afl")
  }

  log.Printf("[%v] running in AFL, shmid: %v", pid, shmname)
  shmid, err := strconv.Atoi(shmname)
  if err != nil {
    log.Printf("[%v] cannot convert shmname %q to integer: %v", pid, shmname, err)
    return nil, fmt.Errorf("error converting __AFL_SHM_ID %q to integer: %v", shmname, err)
  }

  shmptr, _, errno = unix.Syscall(unix.SYS_SHMAT, uintptr(shmid), 0, 0)
  if errno != 0 {
    log.Printf("[%v] failed to access shared memory: %v", errno.Error())
    return nil, fmt.Errorf("cannot access shared memory: %v", errno.Error())
  }

  log.Printf("[%v] shm at 0x%x, converting to accessible slice", pid, shmptr)

  shmHdr = (*reflect.SliceHeader)(unsafe.Pointer(&shared_mem))
  shmHdr.Data = shmptr
  shmHdr.Len = SHM_SIZE
  shmHdr.Cap = SHM_SIZE

  log.Printf("[%v] shm attached, shm[3]: %x", pid, shared_mem[3])
  log.Printf("[%v] shmptr: %x, shmHdr: %p, shared_mem: %p", pid, shmptr, shmHdr, &shared_mem)

  return shared_mem, nil
}
```

Code B.6: Library for attaching **AFL**'s shared memory

```go
package aflshm

import (
  "fmt"
  "io"
  "log"
  "math/rand"
  "os"
  "time"

  "golang.org/x/sys/unix"
)

const (
  FORKSRV_READ_FD  = 198
  FORKSRV_WRITE_FD = 199
)

type Status int32
type PID int32

func AttachForkServer(logger io.Writer, shared_mem SharedMem, fuzzFunc func(SharedMem, io.Reader)
     Status) error {
  log.SetOutput(logger)

  pid, _, _ := unix.Syscall(unix.SYS_GETPID, 0, 0, 0)

  log.Printf("[%v] setting up fork server", pid)
  log.Printf("[%v] seeding random generator for PIDs", pid)
  rand.Seed(time.Now().UnixNano())

  reader := os.NewFile(FORKSRV_READ_FD, "reader")
  writer := os.NewFile(FORKSRV_WRITE_FD, "writer")

  status := int32(0)
  mockPID := int32(0)

  n, err := writeInt32(writer, status)
  if n != 4 || err != nil {
    log.Printf("[%v] (%d/4) cannot write status %d: %v", pid, n, status, err)
    return fmt.Errorf("cannot write to forkserver: %v", err)
  }
  log.Printf("[%v] written status: %d (%0x)", pid, status, status)

  log.Printf("[%v] fork server connected", pid)
  for {
    n, err = readInt32(reader, &status)
    if err != nil && err == io.EOF {
      time.Sleep(1 * time.Millisecond)
      continue
    } else if err != nil {
      log.Printf("[%v] got error reading status: %v", pid, err)
    }

    // use the forkservers pid
    mockPID = int32(2)

    n, err = writeInt32(writer, mockPID)
    if n != 4 || err != nil {
      log.Printf("[%v] (%d/4) cannot write mockPID %d: %v", pid, n, mockPID, err)
      return fmt.Errorf("cannot write PID to afl: %v", err)
    }
    shared_mem[0]++

    status = int32(fuzzFunc(shared_mem, os.Stdin))

    log.Printf("[%v] (for) statusCh returned: %v", pid, status)

    n, err = writeInt32(writer, status)
    if n != 4 || err != nil {
      log.Printf("[%v] (%d/4) cannot write status: %v", pid, n, err)
      return fmt.Errorf("cannot write status to afl: %v", err)
    }
  }
}

func writeInt32(w io.Writer, v int32) (n int, err error) {
  b := make([]byte, 4)
  b[0] = byte(v >> 24)
  b[1] = byte(v >> 16)
  b[2] = byte(v >> 8)
  b[3] = byte(v)
  return w.Write(b)
}

func readInt32(r io.Reader, v *int32) (n int, err error) {
  b := make([]byte, 4)
  n, err = r.Read(b)
  if err != nil {
    return n, err
  }
  *v = (int32(b[0]) << 24) | (int32(b[1]) << 16) | (int32(b[2]) << 8) | (int32(b[0]))
  return n, err
}
```

Code B.7: Library for setting up communication with **AFL**'s fork server

B.3.1 Smartcard communication library in Golang

Code B.8 contains library code for communicating with a smartcard, based on the PC/SC client implementation of [13]. It waits until a card is inserted, selects the provided applet ID, and checks the applet could be selected. Through the `Process` function, a terminal program can send **APDU**s, optionally filtering 'bad' **APDU**s from reaching the smartcard, and receive status words.

```
package passwordeq

import (
  "fmt"
  "log"
  "sort"
  "strings"
```

```go
  "sync"
  "sync/atomic"
  "time"

  "github.com/sf1/go-card/smartcard"
)

type SmartCard struct {
  mut  sync.Mutex
  Ctx  *smartcard.Context
  Card *smartcard.Card

  once          sync.Once
  FirstAPDUSent int64 // time.Now().UnixNano()

  // updated via sync/atomic
  TotalAPDUSent int64
  TotalAPDURcvd int64
  FinalAPDURcvd int64 // time.Now().UnixNano()

  prefixes    *PrefixList
  isWhiteList bool
}

func NewSmartCard(withWhitelist bool, appletID []byte) (*SmartCard, error) {
  prefixes := NewPrefixList(withWhitelist)
  ctx, err := smartcard.EstablishContext()
  if err != nil {
    return nil, err
  }

  log.Printf("waiting for smartcard")
  reader, err := ctx.WaitForCardPresent()
  if err != nil {
    return nil, err
  }
  log.Printf("smartcard inserted")

  card, err := reader.Connect()
  if err != nil {
    return nil, err
  }
  log.Printf("smartcard ATR: %v", card.ATR())

  if appletID != nil {
      log.Printf("selecting applet ID %x", appletID)
    selectAPDU := smartcard.SelectCommand(appletID...)
    response, err := card.TransmitAPDU(selectAPDU)
    if err != nil {
      return nil, err
    }
    if response.SW() != uint16(0x9000) {
      return nil, fmt.Errorf("expected 0x9000 response when selecting applet, got: %s\n", response)
    }
  }

  sc := &SmartCard{
    Ctx:      ctx,
    Card:     card,
    prefixes: prefixes,
  }
  return sc, nil
}

func (sc *SmartCard) Process(apdu []byte) uint16 {
  // Protect the smartcard from random weird APDUs.
  if len(apdu) < 2 {
    return SW_CONDITIONS_NOT_SATISFIED
  }

  if len(apdu) > 261 {
    // Cardman 5121 + libccid supports up to 271 bytes including 10 bytes CCID header.
    // See compatibility matrix <https://ccid.apdu.fr/ccid/section.html#392>
    // See description of dwMaxCCIDMessageLength <https://ludovicrousseau.blogspot.com/2014/11/ccid
     -descriptor-statistics.html>
    // See USB descriptor info <https://ccid.apdu.fr/ccid/readers/CardMan5121.txt>
    return SW_CONDITIONS_NOT_SATISFIED
  }

  if !sc.prefixes.Allowed(apdu) {
    log.Printf("apdu %x not allowed by prefix filter", apdu)
    return SW_CONDITIONS_NOT_SATISFIED
  }

  sc.mut.Lock()
  defer sc.mut.Unlock()
  cmdAPDU := smartcard.CommandAPDU(apdu)
  log.Printf("sending command APDU: %s (raw: %02X)", cmdAPDU.String(), apdu)
  sc.once.Do(func() {
    // We already have the mutex, so we can set time of first APDU now
    sc.FirstAPDUSent = time.Now().UnixNano()
  })
  response, err := sc.Card.TransmitAPDU(cmdAPDU)
  atomic.AddInt64(&sc.TotalAPDUSent, 1)
  if err != nil {
    log.Printf("error transmitting APDU %s (raw: %02X): %v", cmdAPDU.String(), apdu, err)
    if strings.Contains(err.Error(), "broken pipe") {
      panic(fmt.Sprintf("broken pipe from pcscd, input: %x, err: %v", apdu, err))
    }
    return SW_UNKNOWN
  }
  atomic.AddInt64(&sc.TotalAPDURcvd, 1)
  log.Printf("got response APDU: %s (raw: %02X)", response.String(), []byte(response))
  sc.FinalAPDURcvd = time.Now().UnixNano()

  return response.SW()
```

```
}

func (sc *SmartCard) AddPrefix(prefix []byte) {
  sc.prefixes.Add(prefix)
  sort.Sort(sc.prefixes)
}
```

Code B.8: Library for communicating with a smartcard

B.3.2 Smartcard fuzzing program in Golang, fuzzing helper script

Code B.9 provides a commandline interface (CLI) program that connects **`AFL`** to a smartcard, in order to let **`AFL`** fuzz an applet on the smartcard.

```
package main

import (
  "bytes"
  "encoding/hex"
  "flag"
  "fmt"
  "io"
  "io/ioutil"
  "log"
  "os"
  "sync/atomic"
  "time"

  "passwordeq"
  "passwordeq/aflshm"
)

var (
  fuzz      = flag.Bool("fuzz", false, "setup shared memory and forkserver for AFL")
  pcsc      = flag.Bool("pcsc", false, "send commands to the first available smartcard")
  safe      = flag.Bool("safe", true, "if true: use INS_TRY_PASSWORD_SAFE, _UNSAFE otherwise")
  stdinHex  = flag.Bool("stdinHex", false, "if true, convert hex-encoded input from stdin before
     processing")
  logfile   = flag.String("logfile", "", "log to this file")
  whitelist = flag.String("whitelist", "", "use this file (one prefix, hex encoded per line) as a
     whitelist")
  blacklist = flag.String("blacklist", "", "use this file (one prefix, hex encoded per line) as a
     blacklist")
  applet    = flag.String("applet", "", "select this hex encoded applet id before continuing")

  savedPassword = []byte{0x49, 0x48, 0x47, 0x46, 0x45, 0x44, 0x43, 0x42, 0x41, 0x40}

  checker PasswordChecker
)

func main() {
```

```go
flag.Parse()

if *logfile != "" {
  logger, err := os.Create(*logfile)
  if err != nil {
    log.Printf("cannot use %q as output for logging: %v", *logfile, err)
  } else {
    log.SetOutput(logger)
  }
}

usingWhitelist := false
listFile := ""
if *whitelist != "" && *blacklist != "" {
  log.Fatalf("cannot have both blacklist and whitelist")
}
if *whitelist != "" {
  usingWhitelist = true
  listFile = *whitelist
}
if *blacklist != "" {
  listFile = *blacklist
}

prefixList := readPrefixList(listFile)
var sc *passwordeq.SmartCard
var err error
if *pcsc {
  log.Printf("setting up smartcard communication")
  var appletID []byte
  if *applet != "" {
    // Read in hex encoded applet id
    appletID, err = hex.DecodeString(*applet)
    if err != nil {
      log.Fatalf("cannot decode applet ID %q: %v", *applet, err)
    }
  }

  sc, err = passwordeq.NewSmartCard(usingWhitelist, appletID)
  if err != nil {
    log.Fatalf("err setting up smartcard communication: %v", err)
  }
  defer sc.Ctx.Release()
  defer sc.Card.Disconnect()

  log.Printf("successfully connected to card")
  if prefixList != nil {
    log.Printf("setting prefixes using list file %q; is whitelist: %v", listFile, usingWhitelist)
    for _, pfx := range prefixList {
      log.Printf("adding prefix %x", pfx)
      sc.AddPrefix(pfx)
    }
  }
```

```go

    // Log statistics every 10 seconds
    go func() {
      ticker := time.NewTicker(10 * time.Second)
      defer ticker.Stop()
      for {
        <-ticker.C
        sent := atomic.LoadInt64(&sc.TotalAPDUSent)
        recv := atomic.LoadInt64(&sc.TotalAPDURcvd)
        timespanNs := atomic.LoadInt64(&sc.FinalAPDURcvd) - atomic.LoadInt64(&sc.FirstAPDUSent)
        duration, _ := time.ParseDuration(fmt.Sprintf("%dns", timespanNs))
        sentRate := float64(sent) / duration.Seconds()
        recvRate := float64(recv) / duration.Seconds()
        log.Printf("==== SMARTCARD STATISTICS: sent/recv, rate: %d/%d (%d), %.2f/%.2f /s, ratio recv/sent: %.4f%% ====", sent, recv, recv-sent, sentRate, recvRate, float64(recv)/float64(sent)*100.0)
      }
    }()
  }

  // Choose between sending it to a smartcard or to our local instance
  if *pcsc {
    checker = sc
    if checker == nil {
      log.Fatalf("could not set checker to smartcard instance", sc)
    }
  } else {
    checker = passwordeq.NewGolangChecker(savedPassword)
  }

  var checkerInput []byte
  if !*fuzz {
    // Read input from stdin
    log.Printf("reading from stdin")
    data, err := ioutil.ReadAll(os.Stdin)
    if err != nil {
      log.Fatalf("could not fully read stdin: %v", err)
    }
    checkerInput = data

    if *stdinHex {
      log.Printf("converting hex-encoded %q into bytes", data)
      hexData, err := hex.DecodeString(string(data))
      if err != nil {
        log.Fatalf("could not convert stdin to hex string: %v", err)
      }
      checkerInput = hexData
    }

    log.Printf("checkerInput: %02X", checkerInput)
    status := checker.Process(checkerInput)
    log.Printf("got status: %x", status)
    os.Exit(0)
  }

  // Setup shared memory
  shm, err := aflshm.AttachSharedMemory(log.Writer())
  if err != nil {
    log.Fatalf("cannot setup shared memory: %v", err)
  }
  log.Printf("shm: %p", &shm)

  err = aflshm.AttachForkServer(log.Writer(), shm, fuzzTimedPasswordEq)
  if err != nil {
    log.Printf("err from attaching forkserver: %v", err)
  }
}

func fuzzPasswordEq(shm aflshm.SharedMem, stdin io.Reader) aflshm.Status {
  input, _ := ioutil.ReadAll(stdin)
  sw := checker.Process(input) // uint16
  shm[sw]++
  return aflshm.Status(int32(sw))
}

func fuzzTimedPasswordEq(shm aflshm.SharedMem, stdin io.Reader) aflshm.Status {
  input, _ := ioutil.ReadAll(stdin)

  // We use the space shm[0x0000]--shm[0x5fff] (24575 decimal) as timing storage. We use 0-255 per
     memory space, yielding 6266625
  // possible 'ticks'. We stay below 0x6000 to not interfere with ISO7816 defined status words for
     'errors'. With a pause of 10ns
  // after each tick, we can span 62666250ns, or 62ms for one command. Otherwise, we panic,
     stopping the entire process.
  var semaphore uint64 = 1
  go func() {
    var incrementsLeft uint8 = 255
    var currentAddress uint16 = 0
    for atomic.LoadUint64(&semaphore) != 0 {
      if incrementsLeft == 0 {
        incrementsLeft = 255
        currentAddress++
      }
      if currentAddress > uint16(0x5fff) {
        panic("currentAddress out of range: 0x5fff")
      }
      shm[currentAddress]++
      incrementsLeft--
      // sleep 10ns
      time.Sleep(10 * time.Nanosecond)
    }
  }()
  sw := checker.Process(input) // uint16
  atomic.StoreUint64(&semaphore, 0)

  shm[sw]++
  return aflshm.Status(int32(sw))
}

func readPrefixList(filename string) [][]byte {
  if filename == "" {
    return nil
  }
  contents, err := ioutil.ReadFile(filename)
  if err != nil {
    panic(fmt.Sprintf("cannot read prefixlist %q: %v", filename, err))
  }
  splitted := bytes.Split(contents, []byte("\n"))
  prefixList := make([][]byte, len(splitted))
  for i, line := range splitted {
    if len(line)%2 != 0 {
      panic(fmt.Sprintf("error on line %d: hex-encoded values not multiple of 2"))
    }
    decodedLen := hex.DecodedLen(len(line))
    prefixList[i] = make([]byte, decodedLen)
    n, err := hex.Decode(prefixList[i], line)
    if err != nil || n != decodedLen {
      panic(fmt.Sprintf("cannot decode line %d (written %d of %d bytes): %v", i, n, decodedLen, err
    ))
    }
  }
  return prefixList
}
```

Code B.9: CLI program for fuzzing a smartcard with **AFL**

Code B.10 provides a wrapper shell script to start fuzzing a smartcard with **AFL** and the CLI program from Code B.9. It does some pre-flight checks, and provides logging of the output of both **AFL** and the CLI program.

```
#!/bin/bash

set -x

FUZZER=$(which afl-fuzz 2>/dev/null)
WHICHRET=$?

if [ ${WHICHRET} != "0" ]; then
    echo "Cannot locate binary afl-fuzz through which: ${FUZZER}"
    if [ ! -f "/usr/bin/afl-fuzz" ]; then
        echo "Did not find afl-fuzz in /usr/bin, exiting"
        exit 1
    fi
    echo "Manually setting FUZZER to /usr/bin/afl-fuzz"
    FUZZER="/usr/bin/afl-fuzz"
fi

if [ ! -d "in_dir" ]; then
```

```
    echo "Please provide an input directory named 'in_dir'"
    exit 1
fi

if [ ! -d "fuzzer_out" ]; then
    echo "Please provide an output directory named 'fuzzer_out'"
    exit 1
fi

if [ ! -d "log" ]; then
    echo "Please provide a logging directory named 'log'"
    exit 1
fi

TARGET="./go-passwordeq-cli"

if [ ! -f ${TARGET} ]; then
    echo "Cannot find target binary ${TARGET}, exiting"
    exit 1
fi

LOGFN=`date +%Y%m%dT%H%M%S-afl-pweq`
echo "Capturing output in 'log/${LOGFN}-{afl,golang}side'..."

killall go-passwordeq-cli

PCSCD=$(pgrep pcscd)

if [ -z ${PCSCD} ]; then
    echo "pcscd not running, exiting"
    exit 1
fi

AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1 AFL_SKIP_BIN_CHECK=1 ${FUZZER} -i in_dir/ -o fuzzer_out/ -m
      2048 -t 5000000 -- ${TARGET} -fuzz -pcsc -logfile "log/${LOGFN}-golangside" "${@}" | tee -a "
     log/${LOGFN}-aflside"
```

Code B.10: Wrapper shell-script for logging output from Code B.9, and calling **AFL**