MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Fuzzing of the OpenSC Project

Bachelor's Thesis

VERONIKA HANULÍKOVÁ

Brno, Spring 2022

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Fuzzing of the OpenSC Project

Bachelor's Thesis

VERONIKA HANULÍKOVÁ

Advisor: Mgr. Antonín Dufka

Department of Computer Systems and Communications

Brno, Spring 2022


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Veronika Hanulíková

Advisor: Mgr. Antonín Dufka

Acknowledgements

I would like to thank my advisor Mgr. Antonín Dufka for guidance and invaluable feedback during the work on my bachelor thesis. Special thanks go to Ing. Jakub Jelen from Red Hat for his mentoring, providing knowledge about smart cards and answering the immense number of my questions. Finally, I wish to express gratitude to my family and friends who supported me during my studies and writing this thesis.

Abstract

The OpenSC project provides middleware and tools for using smart cards. Smart cards are secure hardware that allows to securely store the data and cryptographic keys and perform cryptographic operations with it. It is important to ensure that OpenSC correctly handles data it receives from smart cards, and fuzzing can be used for this purpose. Fuzzing is a suitable method for testing and detecting errors in code that may affect the application's security. This thesis aims to analyze the state of support for fuzzing in the OpenSC project and propose changes leading to the overall improvement of test results.

Keywords

fuzzing, testing, smart card, OSS-Fuzz, OpenSC

Contents

List of Tables

List of Figures

Introduction

Cryptography is a science concerning securing data from potential attackers [1]. Whether it is data encryption, with a purpose to hide the original data, or signing the data for authentication of the sender, it is necessary to store cryptographic keys for the given operations. Several risks are involved in storing such cryptographic objects on traditional storage media. Storage media are available to other applications, including malicious programs that may misuse the stored data. Insufficient storage security can therefore expose keys and thus compromise the security of confidential data. This issue is addressed by secure hardware.

Secure hardware incorporates computing devices that provide secure storage for sensitive data. In addition, these devices can directly perform cryptographic operations in a secure environment without revealing secret keys. [2]

Smart cards are secure hardware found in many application areas, such as authentication tokens or bank cards. Smart cards provide secure storage of cryptographic keys and other data. Moreover, they can safely perform security-critical operations with secret keys themselves [3]. Smart cards also need to communicate with a host device. Therefore, it is necessary to ensure the secure handling of both the communication with the card and subsequent processing of the received data.

The methods of communication with cards are specified by several standards, some of which are implemented within the OpenSC project [4]. OpenSC provides a smart card library, middleware and tools for personalisation and communication with smart cards [5]. OpenSC API and tools communicate with connected smart cards and process the received data further. Possible flaws in OpenSC code may result in vulnerabilities affecting other applications. Sufficient testing is necessary to ensure security and eliminate such vulnerabilities. Proper implementation of tests can lead to discovering new bugs and reduce the risk of potential threats that may affect users and their data. An example of such a suitable testing method is fuzzing.

Fuzzing is a testing technique that reveals unsafe application behaviour on unexpected or malformed input [6]. It can detect a wide

range of minor bugs that can significantly impact the security of an entire application. Fuzz testing is a convenient method for testing the application since it can simulate untrusted and possibly dangerous inputs from users and cryptographic tokens. OpenSC is already part of the OSS-Fuzz project [7] that provides continuous fuzzing for open-source software. Nevertheless, the fuzzing in OpenSC has only limited support due to the low number of tested operations.

The goal of this thesis is to enhance support for continuous fuzzing of the OpenSC project in OSS-Fuzz. The proposed improvements and new fuzz targets are based on an analysis of the current state of fuzz targets that OpenSC contains. The improvements are also supposed to cover parts that may be in contact with potentially dangerous inputs. In addition to specific design and implementation of improvements and new fuzz targets, the thesis also presents the OpenSC project and its functionality and an overview of fuzz testing as a form of regression testing.

Chapter 1 summarizes basic information about OpenSC project, provided tools and API. The following Chapter 2 presents the fuzzing technique and its possibilities. In Chapter 3, I describe the OSS-Fuzz project and its infrastructure. The next step is analysing the current fuzzing support of the OpenSC project. Chapter 4 contains a description of implemented fuzz targets, an evaluation of statistics and a comparison of the performance of individual fuzz targets. Finally, based on the acquired knowledge, I propose improvements to the current fuzz targets and introduce the implementation of new ones depending on the reports indicating the existing code coverage. An overview of the changes is given in Chapter 5.

During the work on my thesis, the implementation was provided for integration into the OpenSC project so that I could evaluate the performance of my changes directly in the OSS-Fuzz environment. The result of the thesis is the implementation of improvements and partial integration of these changes into the OpenSC project.

1 OpenSC

OpenSC [4] is an open-source project providing a smart card library, middleware, and tools for personalisation and work with smart cards [5]. The project is maintained by the OpenSC team on Github[1]. The project is still active, and a new release is published approximately every year [8].

The Appendix A presents basic information on the operation and communication of smart cards. There is also a description of the PKCS #11 [9] and PKCS #15 [9] standards.

1.1 OpenSC Architecture

The essential components of OpenSC are the library, the code for maintaining the PKCS #15 cards, the code for working with the configuration files, the secure messaging module and the provided tools [4]. Another critical component of OpenSC is PKCS #11 module implementing PKCS #11 API [10].

1.1.1 OpenSC Library

The fundamental part of the OpenSC project is the library `libopensc`. The library contains the implementation of drivers for individual supported cards. Smart card drivers contain functions for card-supported operations, more precisely, sending APDU (Application Protocol Data Unit) commands to operate on the card and processing the received response. Another part of the library is PKCS #15 card emulators providing compatibility with different file structures present on some card types. The library also includes reader drivers for CT-API, OpenCT and PC/SC interface and operations for establishing OpenSC context and general functions enabling card operations and maintenance. [4]

Whether OpenSC supports a smart card is determined by the presence of a driver for the card (and possibly a PKCS #15 emulator). The exact list is given on the project wiki page[2]. Supported cards

1. List of current members can be found at `https://github.com/OpenSC`.
2. OpenSC Wiki is available at `https://github.com/OpenSC/OpenSC/wiki`.

include national ID cards, generic cards representing whole families (with specified models), and USB tokens. [8]

1.1.2 Tools

OpenSC provides several tools for operating with supported smart cards. One of the crucial provided tools is the `pkcs11-tool` for working with PKCS #11 cards. This tool makes it possible to list stored objects, sign, verify, encrypt, and decrypt data. The `pkcs11-tool` requires a module that implements the PKCS #11 API. This module can be linked statically or dynamically in the form of a shared object. [11]

To create the PKCS #15 file structure on the card, users can utilise the `pkcs15-init` tool. It allows users to personalise the card, upload keys and certificates or generate new ones. With `pkcs15-tool`, it is possible to manage a file structure on a card. Tool `pkcs15-crypt` serves for cryptographic operations. [11]

Other tools are used to manipulate and display information about a particular card type. The operation differs according to the capabilities of the card. [11]

1.2 Fuzz Testing

Continuous testing of provided tools and functions that handle potentially dangerous input is an ideal target for fuzzing. Support for fuzzing was added in 2019 [4, pull request 1697]. The OSS-Fuzz project was chosen as a suitable candidate for fuzzing of the OpenSC, mainly due to the possibility of continuous integration of fuzzing for testing. In addition to OSS-Fuzz, fuzzing in OpenSC has also become a part of GitHub actions using CIFuzz [12] concerning new changes in the repository [4, pull request 2130].

Since then, only slight changes have been implemented in existing fuzz targets, mainly correcting minor problems in source code. However, fuzz testing has already revealed many bugs in the OpenSC project. The results show that fuzzing is a suitable way to find hard-to-find problems that can affect the safety of the final product.

2 Fuzzing

Software security is a process that addresses how software can be protected against vulnerabilities that may affect the system's operation. The goal of software security is to prevent attacks that can compromise a given system. It covers several areas that permeate the entire software development. These include secure development practices, issue maintenance, code review and testing. [6]

An integral part of the testing process is static and dynamic code analysis. While the static analysis does not execute the code, dynamic analysis involves testing performed at run time. One of the dynamic analysis techniques and automatic code testing that has been widely used recently is fuzzing. [6]

This chapter defines fuzzing as a testing technique, describes the fuzz testing process and introduces a brief classification of fuzzers. Finally, it clarifies metrics for evaluation of fuzzing performance and fundamental categories of errors discovered by fuzz testing.

2.1 Definition of Fuzzing

Fuzzing is a negative testing technique. Negative testing is characterised by utilising unexpected or malformed inputs sent to the system under test. The fuzzing approach differs from other test techniques (such as performance testing) in the intentionally malformed input type. [6]

The goal is to create an input that causes the tested target to behave unexpectedly and lead to an error. Testing is successful if the input causes the system to crash or force unexpected behaviour. A crash on such an input indicates an issue that can cause erroneous behaviour or even a real crash when the system is deployed. Fuzzing is usually a black-box or grey-box testing approach, where not all internally running operations are known when delivering inputs. The purpose of fuzzing is to generate unexpected inputs and subsequently discover flaws related to security or service availability. [6]

Fuzzers[1] are programs that cover the entire fuzz testing process – creating inputs, supplying them to the system under the test, monitoring the running of the system and reporting problems [6]. The tested interface or, more specifically, a function that uses fuzzing input for testing, is referred to as a *fuzz target* in this thesis[2].

Fuzzing is a relatively simple way of testing that can be well automated. Generating various damaged inputs can better cover the tested program, as it can develop cases that human testers miss. In addition, false positives are rarely found when fuzzing, as only crashes that occurred are reported. [15]

2.1.1 The Process of Fuzzing

The first stage of the testing requires the determination of a suitable target, which can be examined. The target can be, for example, an API or an application, which malformed inputs could compromise. Such targets include security-related applications, applications working over the network, and programs working with higher privileges or confidential data. [16]

It is also necessary to determine how the attacker can disrupt the target's run. Therefore, the second phase is evaluating critical parts of the application. This can include function input arguments, used protocols, environment variables, or memory. [16]

After the initial preparation, the fuzzer can generate a new fuzzing input data or mutate existing inputs, depending on the fuzzer type and its generation technique. The created inputs are passed to the tested interface. [16]

In the next phase, the data are fed as inputs to the tested object, and the tested target runs on the created data. The fuzzer then monitors the run and evaluates encountered problems and crashes. During fuzzing, it is essential to monitor the program's visible behaviour (such as direct crashes) – fuzzer can use tools or components to detect other problems. It can be, for example, responsiveness by protocol implementations or memory corruption [16]. The level of monitoring depends on the specific fuzzer.

---

1. The *fuzzer* is a commonly overloaded term; therefore, OSS-Fuzz and ClusterFuzz use their defined terms [13].
2. The term *fuzz target* is also used by some fuzzers as by libFuzzer [14].

If the fuzzer finds a vulnerability, it can generate output suitable for further investigation, reproduction of the error or other statistics.

2.2 Fuzzer Classification

Due to the diversity of approaches to fuzzing and the goal of testing, it is impossible to divide fuzzers according to only one criterion into specific groups. This section presents multiple possibilities for dividing fuzzers and their types in a given category.

2.2.1 Based on Testing Purpose

Individual types of fuzzers are divided mainly according to the purpose of use.

Single-Use Fuzzers These are fuzzers targeting one particular application or task. They can be quickly created and are used for easy testing. Single-use fuzzers are not generic and cannot be easily extended for further use. [6]

Fuzzing Libraries and Frameworks Fuzzers can also be implemented as libraries that allow reusing for various purposes. These libraries provide an API that is used to test a specific object. Fuzzing libraries belong to generic fuzzers that can test various independent interfaces. Based on their knowledge of the tested object and input structure, they adaptively generate inputs for the tested object to reach new states in the program's execution. [6]

Protocol-Specific Fuzzers Protocol fuzzers are developed for fuzzing various implementations of a given protocol. These fuzzers should follow basic protocol structures. Adapting the fuzzer to a given protocol type increases code coverage. [6]

Commonly tested protocols include, for example, SSL and TLS [6]. Recently developed protocol-specific fuzzer is project `tlsfuzzer` [17].

In-Memory Fuzzers Another option is an in-memory fuzzer. It modifies the arguments in the memory of the tested program before other

functions process them. The tested functions can also be internal functions that are not directly run with user input. [6]

2.2.2 Based on Data Generation

Fuzzers can be categorised based on the process of creating new inputs. Two main approaches are mutating inputs and generating inputs.

Mutation-Based Fuzzers Fuzzers based on generating new inputs with mutations start with a set of valid inputs. The initial set is often referred to as corpus. Files in the corpus are then mutated using (predefined) rules to create anomalies. These rules include, for example, utterly random byte changes or bit-flipping. [6]

Generation-Based Fuzzers This approach does not require an initial set of inputs, as it generates new inputs without using the previous ones. For the fuzzer to work efficiently, it must know the structure of the data it is to generate. A generation-based fuzzer can work well in generating inputs for structured protocols, provided it is familiar with these structures. [6]

2.2.3 Based on Interface Knowledge

Another criterion for fuzzer classification is its knowledge of the structure of generated inputs. This is a different approach to input generation – it depends on how much knowledge the fuzzer needs to have about the tested interface. For an ideal approach, it is valuable to combine the following two mentioned methods, thus adhering to the necessary structure with anomalies. [6]

Intelligent Fuzzer In general, the fuzzer knows the object under test and the structure of the inputs it works with. Newly generated inputs adhere to the given structure. However, a fully smart fuzzer has a low chance of finding a bug since it is impossible to create input causing a crash. [6]

Dumb (Non-Intelligent) Fuzzer The fuzzer generates data mostly at random. That might cause direct rejection of an entry that does not follow the structure in the early stages of testing. The result is low code coverage and a low probability of finding a problem. Therefore, this type of fuzzer is not suitable for testing interfaces that are sensitive to the structure form of input. [6]

2.2.4 Based on the Access to the Tested Object

Just as testing approaches can be divided into white-box, grey-box and black-box testing, this division also applies to fuzzers. The individual categories specify the degree of access of the fuzzer to the tested object.

White-Box Testing In this case, the fuzzer has full access to the source code of the tested object. This is also the only way to achieve higher code coverage. These fuzzers use tracking parts of the code to create new inputs. [6]

Black-Box Testing Object source code is not available with this method. The generation of new inputs by the fuzzer depends only on the object's response to the given input (return value or other program output) and not on any metrics working with the code, e.g. code coverage. [6]

Grey-Box Testing The more efficient work of the fuzzer requires the evaluation of metrics depending on the knowledge of the code. These metrics include, for example, code coverage (of which coverage-guided fuzzers). This knowledge then allows the adaptation of the fuzzer for subsequently generated inputs. Grey-box testing is, therefore, a hybrid approach to the tested object [6]. It is possible to support grey-box fuzzers with a sufficient corpus and a dictionary.

2.2.5 Based on Tested Interface

Fuzzers can be classified according to the type of tested interface. The following section is a brief overview of possible interface targets for fuzzing.

Local Program The fuzzer specializes in the execution of the program in the local environment (as opposed to, for example, fuzzing network communication). Fuzzing inputs contain environment variables, command-line arguments or malformed file formats. Deformed inputs can make the tested program crash unexpectedly and cause memory corruption and possible security threats. File format testing is suitable for various parsers, from multimedia to antivirus gateways. [6]

API Fuzzing Fuzzer supplies inputs directly to functions from API provided by the tested object. Generated parameters become malformed arguments of the tested function [6]. The goal is to cause a crash in the API backend. Fuzzer should know the argument structure of the tested API.

Network Protocol Fuzzing It is possible to test communication with the target over a communicating channel and fuzz protocols used in networking. Sent malformed packets are monitored to determine whether they cause a crash. This fuzzer type may be sensitive to inputs non-compliant with protocol format. [6]

Web Fuzzing That fuzzing type can also be a specification of network protocol fuzzing. It focuses mainly on fuzzing the HTTP protocol. It can also include fuzzing a web application by pushing malformed inputs to forms and testing SQL injections [3]. [6]

Table 2.1 describes the primary classifications of some well-known fuzzers. In particular, the category of access to the tested object may differ based on the specific use.

3. Unchecked input can result in the insertion of a foreign code modifying an SQL statement.

Table 2.1: Classification of selected fuzzers [18].

Fuzzer	Purpose	New data	Knowledge	Access
SPIKE	framework	generation	intelligent	white-box
Peach	framework	generation	intelligent	white-box
AFL	library	mutation	intelligent	grey-box
libFuzzer	library	mutation	intelligent	black-box[a]

a. According to some sources, libFuzzer is referred to as a grey-box fuzzer. The access to the tested object depends on the specific use.

2.3 Evaluation of Fuzzing Efficiency

Several essential characteristics are used to measure fuzzing efficiency. The importance of individual metrics depends on the specific fuzzer type and its use.

Execution Speed The primary metric for performance comparison is the speed of testing by which is meant the speed at which just one test is performed. The execution speed of fuzzing usually depends on the program or function being tested. Slowdowns can occur, for example, when working with memory excessively. Generally, faster execution allows for the testing of more inputs, and it is recommended to aim for a higher test execution speed of around 1000 executions per second. [19]

Code Coverage One of the most fundamental metrics of any testing is code coverage. It is the amount of code of the tested object that was run during testing among all code that can be executed [6]. Different inputs trigger different conditional paths through the executed code and may encounter hidden flaws.

The unit in which code coverage is measured depends on the specific fuzzer – it can be line coverage, function coverage, executed tools, or other regions determined by the abstraction layer [20]. This metric also depends on the code style and concrete implementation used in the tested program.

The coverage tells how much of the code was executed during the testing. Therefore, it is a good metric for displaying the possible range of a fuzzer, and the tester's goal should be to achieve the highest possible coverage (not necessarily 100%). However, there can be internal parts of code that cannot be executed even with a wide variety of inputs [21], or on the contrary, neither higher values may not always mean finding more problems (since the inputs covering the region may not include ones that cause an actual crash) [6]. Some parts of the code are more error-prone than others, so it is sometimes more efficient to include more possible inputs than to cover code that may not cause errors.

Input Space Coverage As mentioned for code coverage, a critical fuzz metric is also the coverage of possible inputs of the tested program. This metric shows which possible inputs were tested and which fuzzer did not reach [6]. The fuzzer should be able to insert anomalies in the appropriate places of the generated input.

For mutation-based fuzzers, suitable input space coverage can be reached with well-defined corpora. Corpus files should adhere to the appropriate input structure.

2.4 Software Bugs

Fuzz testing aims to find malformed inputs that cause errors in the system under test. The most common error categories found when fuzzing are memory-related vulnerabilities, denial of service and data injection vulnerabilities.

Memory-Related Vulnerabilities A possible vulnerability that can arise from bugs in code happens in memory. This can be caused by both reading and writing.

The program can read the memory where it should not have access. In this case, it is possible to reveal some internal information stored in this part of memory. It is also possible that the program writes outside its reserved memory. Then the application data may change. This vulnerability could be exploited by an attacker to change the

execution of a given program or to run its code [6]. This corruption can affect both stack and heap memory.

This vulnerability category also includes other problems, for example, an integer overflow.

Denial of Service The attack aims to limit the system's functionality or degrade its computing power. Users are no longer able to use all of the system resources. This severe security issue prevents users from working with the system. [6]

Data Injection Vulnerabilities In this attack approach, a foreign data is injected into the process performed by the system. One of the methods of data injection is SQL injection, where an attacker inserts a string into the SQL query, which causes an unsolicited operation. A system unprepared for such an attack may face a compromise on its database. [6]

Other attacks may also involve HTTP requests or XML data [6]. Since fuzzing supplies the system with malformed inputs, fuzz testing can reveal vulnerabilities connected with injection.

3 OSS-Fuzz Fuzzing Platform

OSS-Fuzz launched in 2016 as a service for open-source software fuzzing, previously used for fuzzing Chrome components [22]. It works with libFuzzer [14], AFL++ [23] (previously supported AFL [24] was replaced with AFL++) and Honggfuzz [25] fuzzing engines with LLVM sanitizers [26] for continuous fuzzing of projects written in C, Java, Python and other common languages [27].

OSS-Fuzz is widely used. By 2021, it has already discovered over 36,000 bugs in more 500 open-source projects[1] [27]. Among the involved projects, it is possible to find well-known open-source projects such as OpenSSL or GnuTLS [7].

This chapter deals with the operation of the OSS-Fuzz project and its fuzzing environment ClusterFuzz. The fuzzing engines and sanitizers used by ClusterFuzz are also described here.

OSS-Fuzz uses specific terms concerning fuzz testing. The *fuzzing engine* is the program responsible for performing the entire fuzz testing process. In contrast, the *fuzzer* only generates new fuzzing inputs. [13]

3.1 OSS-Fuzz Workflow

Continuous fuzzing in OSS-Fuzz includes several phases - building fuzz targets, the testing itself and reporting bugs. In the first step, OSS-Fuzz downloads the source code from the upstream repository and OSS-Fuzz repository on GitHub [28]. The upstream directory usually contains the particular fuzz targets. However, some projects store fuzz targets directly in the OSS-Fuzz repository [7].

Then, a container is created using base images. The builder[2] builds the project with fuzzing targets according to the specified configuration. It also uploads fuzz targets into cloud storage. [28]

After that, ClusterFuzz [29] uses the compiled fuzz target binaries and performs fuzzing in an isolated environment. The crashes are au-

1. List of public issues can be found at `https://bugs.chromium.org/p/oss-fuzz/issues/list`.
2. The builder details can be found at `https://github.com/google/oss-fuzz/tree/master/infra/build`.

tomatically reported to the issue tracker[3]. ClusterFuzz notifies project members about arising issues and tracks fixes of found crashes. [31]

The overall process is shown in Figure 3.1.

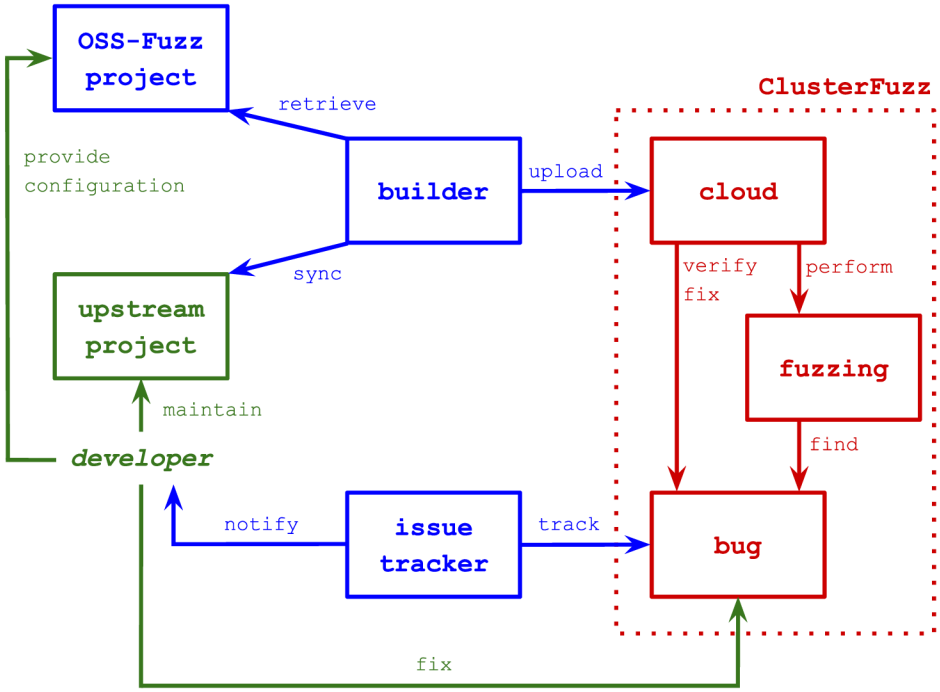

Figure 3.1: OSS-Fuzz workflow adapted from OSS-Fuzz [28] and ClusterFuzz [31] architecture overview schemes.

3.1.1 Integration of the Project

OSS-Fuzz integrates open-source projects at the request of their maintainers. The OSS-Fuzz repository contains all currently integrated projects in the `projects/` directory. Each project's directory includes the project's metadata, build script, and Dockerfile that defines the container environment for building the particular project. [32]

The fuzz targets themselves should be located in the repository of the tested project. This allows faster code changes and the addi-

3. ClusterFuzz uses Monorail [30], which is a Cloud-based issue tracker also used for Chromium projects.

tion of new targets. The following code snippet presents the function signature required for fuzz targets in OSS-Fuzz. [32]

```
int LLVMFuzzerTestOneInput(const uint8_t *Data,
                           size_t         Size);
```

In addition to the fuzz targets themselves, attaching a corpus with files used to generate new entries and dictionaries is possible. The dictionaries contain terms that the fuzz engine can use to generate new inputs. [32]

Dockerfile defines the container environment for the build of the project and its fuzz targets. It must include commands for installing dependencies needed for building the project repository. The build script also runs in this container. It is recommended to link the project statically. After the fuzz targets are built, they are executed in a different environment than the build environment. [32]

Build script contains the basic configuration for building binaries, loading the final fuzz target binaries, and creating ZIP files with corpora and dictionaries in `$OUT` destination. Environment variable `$OUT` indicates where the binaries and build artefacts for fuzzing run are stored. Since the build and runtime environments are different, it is recommended to use static linking. [32]

File `project.yaml` stores the project's metadata. In addition to setting basic information such as homepage, programming language, and the repository, it is possible to configure sanitizers, architecture, fuzzing engine, or the number of builds per day. The configuration file also lists the project members used to control access control to the reported bugs, test cases, and the whole OSS-Fuzz application. [32]

Establishing the project directory in the OSS-Fuzz repository serves as registration in the OSS-Fuzz system, and OSS-Fuzz can start with fuzzing. Alternatively, it is possible to use `infra/helper.py` script for building and running fuzzing locally in a container for local testing purposes. This allows users to reproduce crashes and generate code coverage based on the current corpus. [32]

3.2 ClusterFuzz

ClusterFuzz is a fuzzing backend running on the Google Cloud Platform [31], and it is used by OSS-Fuzz[4]. The infrastructure mediates fuzzing via available fuzzing engines, operates with test cases, and takes care of bug issue management. For a simple overview of the status of fuzz targets and current crashes, ClusterFuzz provides a user-friendly web interface and issue tracker for discovered problems. [29]

3.2.1 ClusterFuzz Architecture

ClusterFuzz runs in the cloud environment and consists of two main components. The first component is App Engine [31] takes care of web interface and crash management. The second component is a pool of fuzzing bots [31] – these work on the fuzzing itself, checking for a crash fix, minimising test cases and pruning the corpus[5].

Job definition characterises the build specification of running one fuzz target with settings for the fuzzing engine and sanitizer [13]. Managing job types is possible on the ClusterFuzz web interface [33].

The workflow starts with uploading builds. According to jobs, selected fuzzing engines with sanitizers perform fuzzing on loaded fuzz target binaries. When a crash occurs, ClusterFuzz sets up issues in the issue tracker and checks daily whether the bug is fixed or not. The period for disclosure is three months from the date of finding the crash. [31]

3.2.2 User and Web Interface

The web interface [34] provides convenient access to outcomes of the fuzzing process. Results are collected daily and presented on several levels for particular fuzz targets and fuzzing engines with sanitizers.

4. ClusterFuzz can be used standalone outside of OSS-Fuzz on Google Cloud or run locally [31].
5. Removal of irrelevant test files without affecting code coverage [13].

Overall Fuzzing Statistics Statistics for selected fuzzing engine in combination with used sanitizer for particular fuzz targets on OSS-Fuzz webpage [35] gather:

- number of found crashes,
- size of the generated corpus,
- average time of test case executions per second and total fuzzing time in hours,
- stability of runs,
- number of newly added tests,
- percentage of a regular crash and startup crash
- percentage of timeouts,
- and average of unwanted log lines.

Data are shown for the selected period. In addition, libFuzzer adds a performance report with fuzz target issues such as low speed or code coverage found while fuzzing and a total number of executed tests. ClusterFuzz stores the metrics mentioned above for every fuzz target from every day. [34]

Code Coverage ClusterFuzz uses Clang [26] compiler with the SanitizerCoverage [36] instrumentation for generating reports. OSS-Fuzz web environment provides a coverage summary of all currently running fuzz targets together [34]. It is possible to generate coverage for individual fuzz targets manually via container from supplied corpus [32].

In OSS-Fuzz, line coverage is the ratio of lines executed at least once during a run to the total sum of executable lines of source code. This is the most straightforward way to track code execution. Region coverage sums coverage of particular code regions[6] executed at least once during fuzz target run. Function coverage delivers the roughest picture since it covers only the execution of individual functions. [20]

Crash Statistics Crash statistics present a list of individual found crashes. It sums up basic information about a particular crash – fuzz target, date, platform, used sanitizer, build details, frequency of crashing, rate of security implication and flag whether the crash is reliably

6. Sections of code divided by control flows or particular code blocks.

reproducible. The webpage[7] also contains a stack trace with sanitizer output and a downloadable test case that might be used for local debugging either with the fuzzing engine itself or in an OSS-Fuzz container. [34]

A security flag appears when a crash is a security-relevant problem [34]. That mainly links to heap and stack buffer-overflows, double frees, write and read operations outside bounds. Flag specifies also estimated severity (unknown/low/medium/high) [35].

Crash overview refers to the issue-tracker [30] issue that is created. Developers have three months to fix the bug before public disclosure [37].

3.3 CIFuzz

In addition to continuous fuzzing through ClusterFuzz, it is possible to use CIFuzz. The CIFuzz is GitHub Actions [38] workflow for fuzzing as part of CI (Continuous Integration) on GitHub. The fuzzing is triggered via pull requests. To use CIFuzz, the project repository must be on GitHub, and the project should be part of OSS-Fuzz. Then, maintainers need to create a workflow in the `.github/` directory of the target project. [12]

3.4 Fuzzing Engines

The concept of a fuzzing engine stands for a tool that creates fuzzed inputs and feeds them to the fuzzing target [13]. Creating new inputs is based upon mutation of already produced data or generation. Particular fuzzer types are described in Chapter 2.

3.4.1 libFuzzer

Library libFuzzer [14] is the only fuzzing engine directly required by OSS-Fuzz [32]. The libFuzzer is coverage-guided and is meant to work with libraries and APIs rather than testing whole programs [14]. For code-coverage instrumentation, libFuzzer uses SanitizerCoverage [36].

7. A list of statistics is available at `https://oss-fuzz.com/` for users who are listed in the project details in the OSS-Fuzz repository.

The fuzz targets are named as `LLVMFuzzerTestOneInput()`, the declaration is presented in the Section 3.1.1. After compilation with Clang and `-fsanitize=fuzzer` flag, Clang builds standalone binaries, which can be executed directly. The libFuzzer also offers many options for running adjustments for running fuzz targets, including memory and time limitations. If specified, it starts fuzzing either from an empty corpus or a user-provided one. The fuzzing engine supports the minimization of corpus files and dictionaries. By default, fuzzing runs until a crash is found or until it is stopped from the command line. [14]

The output contains information describing used corpus files and set options. Further statistics describe at each step current region coverage, corpus size, execution speed, the length limit for entries and memory consumption. [14]

3.4.2 AFL and AFL++

AFL [24] works similarly to libFuzzer; thus, it mutates generated test cases into new ones according to their achieved code coverage. Similarly, it is also possible to define dictionaries used by the fuzzing engine.

AFL++ [23] originated in 2019 as a fork of AFL due to low developer activity. A fuzzing engine combines more mutation techniques and enables custom mutators. It also offers more complex improvements related to corpus pruning [39].

It is possible to use AFL++ and libFuzzer on the same corpus. Moreover, AFL++ can run with the same defined fuzz targets as libFuzzer. Therefore there is no need to create a particular signature in fuzz targets only for this fuzzing engine. Fuzzing with AFL++ does not support corpus pruning and crash minimization. Therefore it is recommended to use libFuzzer as a second fuzzing engine. [40]

3.4.3 Honggfuzz

Honggfuzz has been one of the fuzzing engines used in the OSS-Fuzz project since 2019. It is security-oriented and feedback-driven; it also supports corpus minimization and allows users to generate coverage, use dictionaries and sanitizers. [25]

3.5 Sanitizers

Sanitizers are error detectors and dynamic analyzers of a running program. Their job is to identify problems that could lead to run-time errors. Each sanitizer is specified to recognize a particular class of issues. Common sanitizers include AddressSanitizer [41], UndefinedBehaviorSanitizer [42], ThreadSanitizer [43], MemorySanitizer [44] and LeakSanitizer [45]. This section describes AddressSanitizer with LeakSanitizer, UndefinedBehaviorSanitizer, MemorySanitizer and SanitizerCoverage [36] as they are used in fuzzing with OSS-Fuzz.

Sanitizer libraries have to be linked with the final binary. It is required to use the Clang compiler for its instrumentation and `-fsanitize` flag to specify the sanitizer type while compiling the executable. For most of the offered analysis tools is possible to select either particular checks via the same `-fsanitize` flag or set environment variable to define memory and time limits, listing methods and more [41].

Errors that AddressSanitizer (ASAN) detects are often out-of-bounds accessing, usage after calling `free()` or `return`. OSS-Fuzz uses it together with LeakSanitizer [45] to detect memory leaks. The default configuration of OSS-Fuzz fuzzing use AddressSanitizer with libFuzzer and AFL++ fuzzing engines. [27]

UndefinedBehaviorSanitizer (UBSAN) catches errors connected with out of bound access for static array and bitwise shifts, dereferencing of invalid or `NULL` pointers and overflows for some data types. Other problems found include timeouts. ClusterFuzz also supports UBSAN with libFuzzer fuzzing engine by default. [29]

The only non-default used sanitizer is MemorySanitizer. Its purpose is to find uninitialised reads of values. To utilise this sanitizer, the whole program with all libraries included must be built with MemorySanitizer. Otherwise, the tool will not consider the initialisation of values from the code parts not instrumented from the compilation. [44]

The last supported sanitizer is SanitizerCoverage. Its purpose is to track line, function and region coverage of the executed program's source code. [36]

4 Existing Fuzzing Support in OpenSC Project

Fuzz targets in OpenSC are stored in the `src/tests/fuzzing/` directory along with the corpus directory and Makefile [4]. OpenSC is tested with the libFuzzer fuzzing engine in combination with AddressSanitizer and UndefinedBehaviorSanitizer; the AFL++ fuzzing engine runs with AddressSanitizer [35].

4.1 Existing Fuzz Targets

The following section presents existing fuzz targets implemented in OpenSC before the work on this thesis started.

4.1.1 ASN.1

ASN.1 (Abstract Syntax Notation One) [46] is a data format that enables unified message encoding in communication between different applications. OpenSC provides a basic API for using ASN.1 encoding consisting mainly of encoding and decoding functions and tag processing [4]. These interfaces are tested via two fuzz targets: `fuzz_asn1_print` and `fuzz_asn1_sig_value` [4].

Target `fuzz_asn1_print` calls function `sc_asn1_print_tags()` supplying fuzzing input data and its size. This function prints recursively parsed input buffer according to the ASN.1 format. The fuzz target uses this function as it processes all components of the ASN.1 parsing and thus should cover the majority of the potential issues. [4]

Target `fuzz_asn1_sig_value` aims to test encoding and decoding of ECDSA (Elliptic Curve Digital Signature Algorithm) [47] signatures. The encoding of signature is from `R/S`[1] format into sequence format. The fuzz target first encodes the input fuzzing data and then decodes it into sequence format [4].

4.1.2 Decoding PKCS #15 objects

OpenSC works with the PKCS #15 structure on the card through the internal `sc_pkcs15_card` structure. It also contains a list of structures

1. `R` and `S` are integer values specifing ECDSA signature.

representing the card's actual objects (directory entries). These internal representations can be decoded from the byte sequences. [4]

Fuzz target `fuzz_pkcs15_decode` tests decoding of fuzzing input into internal object structures representing PKCS #15 directory entries. The fuzz target also includes a call to decoding function into public key and token information structures, which are thus tested. [4]

4.1.3 Virtual Reader and PKCS #15 Card Functionality

Smart cards can perform various cryptographic operations like computing signatures or encryption. These operations are invoked via drivers that include functions used to communicate with the card. In the communication between the host and the card, the card driver mediates the creation of APDU requests sent to the card via a reader. The cards respond based on these requests with APDU response packets. [4]

The reader is represented in OpenSC by the `sc_reader_t` structure. The structure contains reader functions, data, and a pointer to the OpenSC context object. [4]

When the reader is to be used for communication with the card, it must be stored in the list of readers that are part of the context structure and be part of the call to the `sc_connect_card()` function, which ensures the connection of the card. If the connection succeeds, APDU requests can be transmitted via a transmit function of the reader structure. [4]

Virtual Reader The fuzz target `fuzz_pkcs15_reader` implements a simple virtual reader that simulates the communication with a virtual card. The reader structure uses fuzzing input data that it interprets as APDU responses. [4]

The reader structure contains `release`, `connect`, `disconnect`, and `transmit` operations. The `connect` operation sets the ATR (Answer To Reset) value of the simulated card, and the `transmit` operation sets the appropriate data in the APDU response. [4]

The ATR and the response data are created via `fuzz_get_chunk()` function, which processes data stored in the virtual reader. The function parses the data into individual chunks whose length is determined by the first two bytes. [4]

The usage of the virtual reader in fuzz target assumes initialised OpenSC context. The first step of creating the virtual reader is to remove all readers stored in the context object. That is followed by the setting reader operations in the reader driver. Next, a `fuzz_add_reader()` function allocates the reader object for the virtual reader. It sets its provided operations and supplies the fuzzing input as reader data. [4]

The `fuzz_pkcs15_reader` fuzz target The fuzz target begins with the initialization of the virtual reader. Then it creates the internal `sc_pkcs15_card` structure and tests decipher, wrap, unwrap, sign and pin operations. [4]

The `fuzz_pkcs15_reader` has a relatively large reach. It is possible to test individual cryptographic operations with it. It also covers card detection and initialisation and functions implemented in individual card drivers.

4.2 Code Coverage

Table 4.1: Coverage of source directory `src/` from 23th October 2021 before integration of new fuzz targets into project [48].

Path	Line	Lines	Region	Regions
`common/`	42.76%	263/615	38.91%	200/514
`libopensc/`	40.88%	26 229/64 168	33.43%	28 741/85 962
`pkcs11/`	0.00%	0/0	0.00%	0/0
`pkcs15init/`	0.00%	0/15 360	0.00%	0/22 627
`scconf/`	10.19%	79/775	7.24%	36/497
`sm/`	0.00%	0/54	0.00%	0/38
`tests/`	95.13%	254/267	95.24%	140/147
`tools/`	0.00%	0/28	0.00%	0/27
`ui/`	25.00%	1/4	25.00%	1/4
Total	33.01%	26 826/81 271	26.52%	29 118/109 816

Figure 4.1 presents the overall line and region code coverage of the OpenSC project. The second and fourth column in the table represents the ratio of the executed code lines (regions) to the lines (regions)

found in some execution paths [48]. There can be seen the highest percentage in `tests/` directory, which is caused by executing the fuzz targets themselves.

All fuzz targets access the `libopensc/` directory primarily. It contains all particular card drivers and PKCS#15 emulators tested mostly by `fuzz_pkcs15_reader`. There are also source files for processing data in ASN.1 format covered by fuzzing targets `fuzz_asn1_print` and `fuzz_asn1_sig_value`.

The report [48] consists of the files, including code parts that are reachable from current fuzz targets. Since it is focused only on listed directories, directories for `minidriver/`, `pkcs11/` and `smm/` are not considered since none of their functionality is reached by fuzz targets.

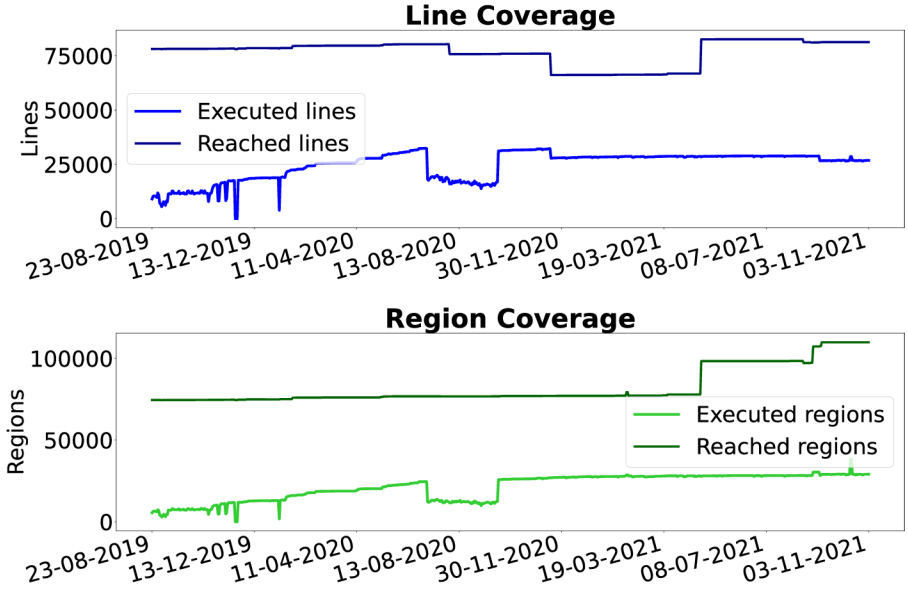

Figure 4.1: Code coverage from 23rd August 2019 to 3rd November 2021 [48].

Figure 4.1 presents overall coverage statistics of line and region coverage. The fluctuations in 2019 are primarily due to the frequent crashes [4]. These have the effect of slowing down the speed and lowering coverage. In 2019, 60 crashes were found. The `fuzz_pkcs15_reader` spent only 21.5% of the run time on fuzzing [35].

In 2020, there was a sharp decline in coverage of almost 20% caused by the fuzz target `fuzz_pkcs15_reader`. Its region coverage dropped from 31.05% to 14.80% [35]. The others did not notice significant deviations. In this period, the coverage of `libopensc/` significantly decreased [49]. There were no significant changes in the OpenSC project, which could have led to this drop [4].

Another significant drop occurred in April 2021, when line coverage fell from 42.95% to 34.44%. The reason was that the fuzz target found a path to the `pkcs15init/` directory, and the total amount of code available increased. [50]

For the overall coverage evaluation, it is essential to compare the statistics of all fuzz targets with each other. The difference between fuzz targets is caused by having a different possible reach in the code. In Table 4.2 can be seen code coverage of `src/` directory per fuzz target.

Table 4.2: Region coverage of source directory src/ from 27th November 2021 by existing fuzz targets [35]. Data are obtained from corpus files generated by OSS-Fuzz [51].

Fuzz target	Region	Regions
`fuzz_asn1_print`	13.73%	383/2,789
`fuzz_asn1_sig_value`	0.70%	770/109 478
`fuzz_pkcs15_decode`	1.37%	1 504/109 480
`fuzz_pkcs15_reader`	25.56%	28 283/110 635

The overall reach of fuzzing in an OpenSC project is limited by the relatively small number of tested functions.

4.3 Performance of Fuzzing Runs

OSS-Fuzz provides further fuzzer statistics and performance reports accessible only to project members [35]. Project members are set via `project.yaml` file in OSS-Fuzz repository [32].

Table 4.3 containing the speed of execution and fuzzing time shows that `fuzz_pkcs15_reader` is the slowest among all. With libFuzzer, it is significantly less than recommended 1000 executions per second [19]

from the two years of fuzzing. The slowdown can be caused by excessive logging or the dynamic allocation in the code.

Table 4.3: Internal performance statistics [35] of libFuzzer engine with ASAN [41] and UBSAN [42].

Year	Fuzz target	ASAN		UBSAN	
		Speed	Time	Speed	Time
2019	fuzz_asn1_print	4 321	92.2%	1 231	16.7%
	fuzz_asn1_sig_value	3 739	91.3%	8 013	99.5%
	fuzz_pkcs15_decode	2 431	14.9%	3 229	11.7%
	fuzz_fuzz_pkcs15_reader	104	21.5%	160	24.1%
2020	fuzz_asn1_print	5 114	99.5%	10 290	99.6%
	fuzz_asn1_sig_value	4 807	99.6%	9 387	99.6%
	fuzz_pkcs15_decode	3 622	99.6%	7 295	99.6%
	fuzz_fuzz_pkcs15_reader	223	42.9%	523	82.6%
2021	fuzz_asn1_print	5 224	95.9%	8 734	95.9%
	fuzz_asn1_sig_value	6 008	95.6%	9 559	95.6%
	fuzz_pkcs15_decode	3 770	95.6%	6 059	95.6%
	fuzz_fuzz_pkcs15_reader	333	73.1%	679	95.5%

Time in the Table 4.3 represents fuzzing time percentage. It is a ratio of total time spent fuzzing. The overall testing time, including states when fuzzing, is interrupted by crashes (either startup and new ones), logging and timeout [35]. On average, `fuzz_pkcs15_reader` achieves less fuzzing time than the rest of the fuzz targets – mainly due to the frequent crashes found by this target [35].

Fuzzing with UBSAN, according to Table 4.3 reports a significantly lower fuzzing percentage in 2019 than in the following years. This was probably caused by frequent crashes encountered by the presented three fuzz targets [35].

4.4 Crash Statistics and Discovered Bugs

The Monorail [30] issue tracker allows direct access to discovered bugs only to project members. However, after fixing the bug or disclosure, the bug reports are opened to public[2].

Fuzzing engine AFL++ with ASAN has run for a shorter period. It has found five crashes only by `fuzz_pkcs15_reader`, from which all were evaluated as security-relevant. Four crashes were caused by timeout, heap-buffer-overflow and subsequent read operation. [35]

The libFuzzer engine has run since 22nd August 2019. The target `fuzz_asn1_print` crashed three times, once on the security-relevant problem with heap-buffer-overflow. The second target `fuzz_asn1_sig_value` has discovered two problems, one of them being security-relevant heap-buffer-overflow. [35]

Target `fuzz_pkcs15_decode` was significantly more successful than the previous two. It has encountered 20 crashes, among which there were bad free callings, heap-buffer-overflows and direct memory leaks and undefined behaviour such as aborts or integer-overflow. [35]

Fuzz target `fuzz_pkcs15_reader` achieves the highest amount of found bugs. It has discovered 117 crashes with ASAN and 39 crashes with UBSAN; 75 were security-relevant. It was most often an out-of-memory approach as heap-buffer-overflow and stack-buffer-overflow. [35]

4.5 Conclusion of Current Fuzzing Support

When comparing all fuzz targets, `fuzz_pkcs15_reader` stands out significantly. Despite the slowest running speed, it exposed the most crashes.

The other fuzz targets run relatively stably and with sufficient speed. Nevertheless, `fuzz_asn1_sig_value` and `fuzz_pkcs15_decode` have low code coverage and have found small number of bugs. To achieve better results, it is necessary to examine the coverage reports for the given fuzz targets and find possible limitations causing the mentioned problems.

2. Crashes described in this section can be found in the OpenSC repository as links to the Monorail reports included in the corresponding commits.

5 Improvements for OpenSC Fuzzing

The following chapter presents the implementation part of this thesis. Several OpenSC parts were selected as suitable for fuzz testing: general card operations, configuration parser, PKCS #15 object encoding, PKCS #11 module, PKCS #15 compatible card initialisation and some of the OpenSC tools. Other changes concern improvements to existing fuzz targets and overall improvement in fuzzing for local regression testing. Each section describes the selected OpenSC part, the structure of a new fuzz target, comments for the implementation, corresponding corpus creation, and discovered bugs[1].

The run of the new fuzz targets was analysed either from the local environment using a container or directly from reports in the OSS-Fuzz project. The fuzz targets were further enhanced according to run statistics and performance.

Newly implemented fuzz targets for encoding PKCS #15 objects, PKCS #15 card personalisation, PKCS #11 API and OpenSC tools are described in this chapter. Further changes regarding the improvements of the two existing fuzz targets, implementations of new fuzz targets for the configuration parser and generic card operations can be found in Appendix B. There is also a future proposal for fuzz target testing `pkcs11-tool` using the SoftHSM [52] software token. The improvements in Appendix B are beyond the scope of the mandatory part of this thesis.

5.1 Encoding of PKCS #15 Objects

OpenSC provides API for encoding PKCS #15 objects from the internal object representation of PKCS #15 structure on the actual card into bytes. The requirement for their usage is initialised `sc_pkcs15_card` object [4]. According to the code coverage obtained from OSS-Fuzz reports [48], these encoding functions are not reachable by any existing fuzz target and thus are not tested.

1. The bugs found by OSS-Fuzz described in this chapter have already been fixed, and their descriptions are available in the corresponding pull requests in the OpenSC repository [4].

5.1.1 Structure of Fuzz Target `fuzz_pkcs15_encode`

The first part of the new fuzz target `fuzz_pkcs15_encode` consists of establishing OpenSC context and connecting the virtual reader. The fuzzing input is interpreted as APDU responses.

Subsequently, the `sc_pkcs15_bind()` function is called. After initialisation, this function sets up the PKCS #15 card structure containing PKCS #15 file objects. The initialisation is crucial because it is impossible to test encoding functionality without the PKCS #15 card object.

The main testing happens in the iteration over file objects. The corresponding encoding function is called based on the type of object. The result is stored in the auxiliary buffer and then released.

In addition to the encoding itself, the conversion of the object to the internal structure of the public key is tested for the public key object. Similarly, the fuzz targets call function for converting a certificate object to an object containing private key attributes.

Finally, a function for encoding the unused space representation is called after the cycle. Allocated resources are released before the termination of the program.

Corpus The input for the fuzz target is only the list of APDUs used by the virtual reader as the card responses. For the PKCS #15 card object to finish the initialisation via virtual reader and work correctly, the input must contain responses related only to the card connection. The corpus files were created from the already existing corpus of fuzz target `fuzz_pkcs15_reader`.

5.1.2 Run Analysis

After 30 days of running in OSS-Fuzz, the fuzz target reaches on average 500 executions per second and has 22.43% region coverage. Coverage is gradually increasing, according to OSS-Fuzz statistics. [35]

5.1.3 Bugs Discovered by `fuzz_pkcs15_encode`

The fuzz target did not encounter any problems during the local or OSS-Fuzz run.

5.2 Personalisation of PKCS #15 Card

Tool `pkcs15-init` tool serves as a utility for the card's personalisation. The card must be compatible with PKCS #15 standard [4]. It allows users to create PKCS #15 structure, add keys and certificates and manage PINs on the card. The relevant source code is stored in `libopensc/` and `pkcs15init/` directory, with 0% code coverage according to OSS-Fuzz statistics [48].

The personalisation is specified via a profile file. The profile defines the PKCS #15 structure layout and contains information about files and objects stored on the card. The `pkcs15init/` directory holds one general profile and other more specific profiles for various types of cards. The profile file structure is similar to the structure of the configuration file, and therefore the same parser is used. Parses profile is required for most of the PKCS #15 initialization functions. [4]

5.2.1 Structure of Fuzz Target `fuzz_pkcs15init`

The fuzz target begins with splitting the fuzzing input data into two main blocks with a null byte as a divider. The first block serves as data of the profile file. The second block is processed APDU responses. The target also uses data from this block as input buffers to test functions. The data are obtained with `fuzz_get_chunk()`. The following steps try to simulate the functions from the `pkcs15-init` tool.

First, the OpenSC context is established, and the virtual reader is created and supplied with fuzzing input data. Connecting the card and transmitting data uses a second input data block to communicate with the card.

After establishing the card connection, the target needs to load the profile. This functionality is emulated since the original function is supplied with a filename and works directly with the profile file stored in the OpenSC directory (fuzz target binaries do not have access to the profile files in the execution environment). That function parses the profile string, processes it, and creates a profile structure with appropriately set attributes. This functionality is similar to the original implementation of the `sc_pkcs15init_bind()` function.

Testing the tool's functionality is split into individual functions that imitate operations in the `pkcs15-init` tool. The fuzz target tests

card initialization, PIN settings, object storage, generation of all possible symmetric and asymmetric keys, symmetric key storage, card finalization and card erasure. These operations take various flags and buffers as arguments. Some of them (such as PIN values) have been selected as static; some are separated from the APDU part of the input file.

It is possible to store the private key in PEM format, the public key in DER or PEM format, and the secret key and certificate. When storing a private and public key, the `pkcs15-init` tool first transforms the content of the input file into a key format from OpenSSL and then converts it into argument structure. Due to this procedure, the target would have to transform the random input into a valid OpenSSL key structure. Therefore private and public key storage testing is omitted in the fuzz target.

Corpus The profile part of the corpus file consists of two files. These are `pkcs15.profile` with the information needed for every card and `myeid.profile` with specifications for the chosen card. In the original `pkcs15-init` tool, there is gradual loading, firstly program parser `pkcs15.profile` and then continues with `myeid.profile`. The corpus file has both profile files directly attached.

The MyEID card was used to generate the APDU traces from the `pkcs15-init` tool. APDU traces are split and connected behind each other to match particular procedures. The exact number of removed bytes is calculated from the log files containing sent and received APDUs created when generating card responses from the `pkcs15-init` tool.

For functionality not supported by the MyEID card, the sections were filled with responses from other parts of the communication. Just enough bytes were truncated so that the answers were enough to complete the function (with an error) so that they did not interfere with the following test block.

5.2.2 Run Analysis

After two month of fuzzing with the `fuzz_pkcs15init`, OSS-Fuzz reports 20.73% of region coverage [35]. It raised from initial 14.09%. The

average speed of test execution is relatively slow; it reached maximally 548 executions per second [35].

The low speed is caused mainly by frequently repeated crashes [35] until bugs are fixed in the code. Another reason may be the complexity of that fuzz target. Using a virtual reader generally slows down fuzzing, as seen in other fuzz targets that use the reader. The `fuzz_pkcs15init` also tests multiple operations consecutively, which can affect the execution time of the test case.

5.2.3 Bugs Discovered by `fuzz_pkcs15init`

Despite the lower speed, the fuzz target is relatively successful in finding bugs. Most cases relate to memory leaks, out-of-bounds reads and timeouts [35].

Local Run with libFuzzer Engine Local run in container showed many small memory leaks in `pkcs15-init` functions and profile parser. Another common bug was dereferencing pointers without checking whether they have a `NULL` value. Subsequent problems arose from reading outside of allocated memory and using variables defined out of the current scope. The initial pull request contains a patch to fix these bugs[2].

OSS-Fuzz Run The `fuzz_pkcs15init` fuzz target encountered more than 20 reproducible problems during the first two months of running in OSS-Fuzz. The issues detected in profile file parsing and functions for PKCS #15 initialisation included `NULL` dereferencing, buffer overflows and memory leaks. [35]

5.2.4 Improvement of the `fuzz_pkcs15init`

Implemented fuzz target `fuzz_pkcs15init` is relatively slow due to a large number of operations. As input, it also takes a complex structure that must cover the profile's content and the APDU responses for many operations. In addition, some of the operations receive buffers with hard-coded values as input instead of fuzzed arguments.

2. Details can be found in original pull request `https://github.com/OpenSC/OpenSC/pull/2500`.

To solve these problems, fuzzing individual operations can be done separately. The first byte of the input is used to select the tested operation. To separate the profile part from the fuzzing input, two bytes specifying the length of this part are taken instead of the null byte divider. All arguments needed for the tested operations are further parsed from the input data.

The described implementation is a proposal for future improvements.

5.3 PKCS #11 API

The PKCS #11 standard defines API for operations accessible on cryptographic devices [9]. Details of this standard are given in Appendix A.4.1. OpenSC project provides a module that implements this interface. The particular implementations of functions defined by the PKCS #11 API are stored in `src/pkcs11/` directory [4].

The primary use of the PKCS #11 API in OpenSC is in the `pkcs11-tool`, which allows performing cryptographic operations with a connected smart card. With default settings, this tool uses the PKCS #11 module from OpenSC. [4]

As shown in the code coverage reports from OSS-Fuzz [48], the PKCS #11 is not covered by any fuzz target.

5.3.1 Structure of Fuzz Target `fuzz_pkcs11`

Fuzz target `fuzz_pkcs11` aims to test PKCS #11 API implementation provided by OpenSC. The PKCS #11 module is linked statically to the fuzz target.

The fuzz target allows testing one procedure at a time. Initially, the first byte of the fuzzed input data decides which procedure to test. For each procedure, the fuzz target then extracts the data needed for testing - PINs, data buffers, and the values used in the templates for key generation, key derivation and key wrapping.

The template is an array that contains attributes that specify information about the type and usage of the new key. First, for each attribute, according to the value of one byte, it is decided whether it

will be part of the template. The following bytes are interpreted as the value of the attribute to be stored in the template.

The PKCS #11 library needs to be initialised to use the API. The `C_Initialize` [10] function is used for this purpose. All connected cards are detected during the initialisation, and virtual slots are created. In `pkcs11-tool`, the slots are further used for card selection for particular procedures.

As the fuzz target does not aim to test real connected cards, it removes any connected readers and deletes the created virtual slots after initialisation. The original list of slots is replaced by only one newly created virtual slot connected to the virtual reader that contains the fuzz input data. This process simulates the detecting the real card in the function `card_detect()`. The original detection function cannot be used because it is impossible to supply the virtual reader.

Fuzz target focuses on testing PIN operations, token initialisation, decryption, signing, verification, key wrapping and key derivation. Some of the mentioned procedures, such as encryption, hashing, signing and signature verification, enable data processing as the whole or per parts. The fuzz target tests both approaches. For sequential processing, it divides the data buffers into smaller parts and calls the operations in a loop.

Corpus It is possible to generate APDU traces directly from running the fuzz target built-in non-fuzzing mode[3]. Since the `C_Initialize` function is called before connecting to the virtual reader, it is only needed to skip deleting the virtual slots and connecting the new reader. Then, the fuzz target can work directly with the connected card. The arguments needed for the given operations are appended before the given APDU traces to create the final corpus of files.

5.3.2 Run Analysis

This fuzz target is not merged in OpenSC yet. Nevertheless, the local test run does not show any performance problems. According to the

3. Via the `fuzzer.c` with `main()` function, that takes a file and supplies its content to the fuzz target as fuzzing input.

corresponding local code coverage report, the created corpus covers the paths leading to the tested functions.

5.3.3 Bugs Discovered by `fuzz_pkcs11`

Since the fuzz target is not integrated into OpenSC, this section describes only bugs found locally while testing the fuzz target.

Local Run with libFuzzer Engine Only one memory leak was found during local `fuzz_pkcs11` fuzz target testing. This issue is caused by the non-releasing of the list of operations part of the PKCS #11 structure for the session. Fixing the problem is part of the initial pull request.

5.4 OpenSC Tools

Essential parts of the OpenSC project are tools provided for working with smart cards. They are located in `src/tools/` directory. Some tools are generic, so users can use them with various smart cards (that comply with PKCS #11 and PKCS #15 standard) or specific for particular card types. [11]

The structures of the tools are quite similar. They always contain a `main()` function that handles command-line options and performs a sequence of operations based on the used options. Tools specialised for a given card type can set a single card driver before attempting to connect the card – only this one is used when connecting. If the user tries to use a non-compatible card, the connection fails, and the tool is terminated. The sequence of the executed operations also depends on whether some form of authentication is required, which is addressed differently by each tool. [4]

The tools in OpenSC are not tested by any fuzz target, as shown by the code coverage report from OSS-Fuzz [48].

5.4.1 Fuzz Targets for Tools

The problem with tool fuzzing is the `main()` function in the tool's source code. It is impossible to link or include the source code directly

since the fuzzing engine includes its `main()` during the testing phase. However, two basic approaches enable fuzz testing of tools.

The first approach is implementing a fuzz target to simulate the tool's operation. That means that besides omitting the processing of options, fuzz target needs to reimplement functions included in the tool source code and call them in the same manner as the tool does in `main()` function.

This approach is quite challenging, as it is essential to rewrite the complete functionality of the tool. In case of significant changes in the tool's source code, adjusting the corresponding fuzz target is also necessary. In addition, some tools have most of the functionality implemented directly in the source file, which puts even more emphasis on proper reimplementation. Nevertheless, this approach is possible, and some tools' functionality is suitable for this type of testing. An example is `fuzz_pkcs15init`, which works on a similar basis as the `pkcs15-init` tool and tests the functionality for PKCS #15 initialisation.

The second approach is to test the tool run in the body of the fuzz target with fuzzed command-line options. To enable direct testing of the tool, it is needed to include the source code and rename the `main()` function[4]. Renaming can be done via `#define` directive, which is pre-processed before the compilation. In the same way, the fuzz target can change the default function for connecting the card to the connection of the virtual reader.

The advantage of this approach is that there is no need to reimplement tool operations. On the other hand, file processing is a bit problematic. Suppose the argument of the input file appears between the fuzzed options. In that case, such a file does not exist in the test environment. The absence of an input file would cause the tool to terminate, and fuzzing would not cover the corresponding parts of the code. Another problem is that the fuzz target does not subsequently delete the output files created during testing.

4. Since it is not possible to assume the current working directory of the fuzz target according to the OSS-Fuzz documentation [27], the tool cannot be called via `system()` (function executing the commands).

5.4.2 Structure of Fuzz Targets for Tools in OpenSC

The implemented fuzz targets for tools in OpenSC projects follow the second approach described in the previous subsection. The renaming procedure is presented in the following code snippet.

```
#define main _main
#define util_connect_card_ex
   fuzz_util_connect_card
#include "tools/pkcs15-tool.c"
#undef main
```

It is possible to call the tested tool with fixed predefined arguments or parse the entire command-line options from the input. The fuzz target first decides which of the mentioned strategies is tested depending on the first byte of the fuzzing input.

The first of these strategies is used to test tool's operations that require an input file[5]. The fuzz target first parses the fuzzing input to get values for command-line options for the tested function of the tool. These values are separated by one null byte as presented in Figure 5.1. Subsequently, it creates a new file stored in the `/tmp` directory. The file size is determined by the two bytes of fuzzing input (referred to as `len1` and `len2` in Figure 5.1) converted into a number. Then the file is filled with data of a given length, following immediately after bytes specifying the size.

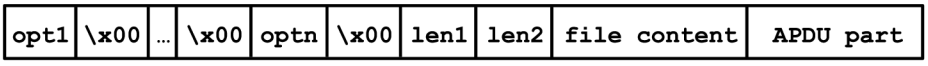

Figure 5.1: Parts of the fuzzing input used for testing with fixed arguments.

The second strategy of interpreting input data parses a part of the fuzzing input as command-line options. It passes them to the renamed `main()` function. The structure of the interpretation of input data is

5. If a file is created during fuzz testing, it should be deleted at the fuzz target's end. However, the file may not be deleted if a crash is found. The creation of files also might slow down test execution. However, tool operations cannot be easily reimplemented to be tested without the usage of a file.

shown in Figure 5.2. The argument part is separated from the APDU part by two null bytes. If any created option requires a file name, this file is not created, and the tool is terminated.

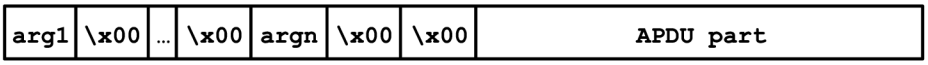

Figure 5.2: Parts of the fuzzing input used for testing with fuzzed arguments.

As shown in the presented scheme in 5.2, two null bytes mark the end of the argument part. Therefore it is not possible to extract from the input an empty argument. This is not a problem that would influence the range of tested inputs, since `getopt_long()` [53] function used in the tools can take arguments in two forms: `-option value` and `-option=value`. It is impossible to generate the first variant as the value cannot be an empty string. However, the fuzz target can create the second variant with an empty value as `-option=`.

Implemented Fuzz Targets for Tools Within the implementation part of this thesis, three fuzz targets for tools were implemented and integrated into the OpenSC upstream: `fuzz_piv_tool`, `fuzz_pkcs15_tool` and `fuzz_pkcs15_crypt`. For the future improvements of fuzzing support in OpenSC, eight fuzz targets were implemented for tools working only with cards of the given type. These tools include, for example, `cryptoflex-tool` or `cardos-tool`.

Tools `pkcs11-tool` and `pkcs15-init` This testing method cannot be applied directly to tools that contain `exit()` functions. If a fuzz target encounters such a call, it crashes as if it found a bug. Therefore, the fuzz target cannot test `pkcs11-tool` and `pkcs15-init` as described, as long as the tools contain an `exit()` calls [4]. Moreover, `pkcs11-tool` contains initialization and connection of the card using the PKCS #11 API, the call of which cannot be easily renamed as is the case with the `main()` function [4]. For the mentioned reasons, fuzz targets testing only the specific PKCS #15 and PKCS #11 functionality are implemented.

Corpus For corpus files, it is possible to use APDU traces generated from original tools directly. This data only needs to be prepended with the values of the arguments appended to the beginning of the corpus file.

5.4.3 Run Analysis

Three fuzz targets testing tools were integrated into the OpenSC project: `fuzz_piv_tool`, `fuzz_pkcs15_tool` and `fuzz_pkcs15_crypt`. The data are evaluated after 18 days of running in OSS-Fuzz.

The target `fuzz_piv_tool` reaches speed over 1000 executions per second on average [35]. Although the region coverage reaches only 3.18% at maximum [35], 69.78% of the available regions of the `piv-tool.c` file are covered according to the total code coverage of the project [54].

Fuzz target for `pkcs15-crypt` achieves higher region coverage around 21.09% [35]. Region code coverage of the tool's source code is 84.31% [54]. The speed is around 400 executions per second [35].

The region coverage of `pkcs15-tool` fuzzed by `fuzz_pkcs15_tool` reaches only 8.84% according to OSS-Fuzz report [54]. This significantly lower coverage is due to a malformed integrated corpus.

5.4.4 Bugs Discovered by Fuzz Targets for Tools

Several issues were found using `fuzz_piv_tool`, `fuzz_pkcs15_tool` and `fuzz_pkcs15_crypt` during both local and OSS-Fuzz fuzzing.

Local Run with libFuzzer An interesting bug discovered during local fuzzing is not resetting the flag from the card structure in the `sc_lock()` function when an error occurs. If the flag remains set from `sc_lock()`, the memory allocated for the card can not be freed.

OSS-Fuzz Run OSS-Fuzz has encountered three bugs[6] so far. They relate to the use of `free()` on an invalid pointer and the associated abortion, `NULL` dereference and a memory leak. [35]

6. Details can be found at `https://github.com/OpenSC/OpenSC/pull/2553`.

5.5 Conclusion of Fuzzing Support Improvements

The described improvements to the existing fuzz targets and the implementation of new ones led to an overall improvement in the state of fuzz testing in the OpenSC project. The new fuzz targets are also designed to cover previously untested code parts.

Existing fuzz target `fuzz_pkcs15_decode` and virtual reader have been modified to cover the tested code better. Changes in the `fuzz_pkcs15_decode` target increase the low code coverage mentioned in Section 4. The modifications mentioned above are described in Appendix B.

Table 5.1: Coverage of OpenSC parts affected by new fuzz targets from 16th May 2022 [54].

Path	Before improvements		After improvements	
	Region	Regions	Region	Regions
`common/`	38.91%	200/514	39.88%	205/514
`libopensc/`	33.43%	28 741/85 962	41.60%	36 645/88 099
`pkcs15init/`	0.00%	0/22 627	24.50%	5 593/22 833
`scconf/`	7.24%	36/497	55.64%	281/505
`tools/`	0.00%	0/27	24.98%	535/2 142

The Figure 5.1 presents region code coverage of OpenSC parts affected by implemented and integrated improvements. The particular pull requests referencing the integration of fuzz targets are described in Appendix C.

The most compelling difference according to Figure 5.1 is coverage of `pkcs15init/` and `scconf/` directory, that show the largest percentage improvement. However, the most significant difference in the number of covered regions is shown by `libopensc/`. Tests have covered more than 7 000 new regions.

Another indicator of success is the number of bugs found. Because of the implemented changes, more than 40 problems have already been detected [35].

Conclusion

The goal of this thesis was to analyse the current state of fuzzing support in the OpenSC project and implement improvements to enhance testing coverage.

This thesis presents the structure and functionality of the OpenSC. It explains the principles of fuzz testing and the possibilities of its use. The thesis also describes the OSS-Fuzz project for continuous fuzzing of open-source software.

The performance of individual fuzz targets that were part of OpenSC was analysed based on reports from the OSS-Fuzz project. It turned out that a great benefit is the implementation of a virtual reader that allows simulation of the communication with the card. Thanks to this, it is possible to test operations implemented in the drivers of individual cards. However, the mentioned fuzz targets tested only a small number of functions, and therefore their reach was limited.

New suitable targets for testing were chosen based on the information obtained from the code coverage of individual parts of the OpenSC project. The implementation of new fuzz targets focuses on configuration file processing functionality, PKCS #15 related operations, PKCS #11 API, generic card operations, and tools provided by OpenSC.

Appendix B describes improvement suggestions beyond the scope of the mandatory part of the thesis. These suggestions include changes for simplification of local regression testing and improvements of the existing fuzz target. There is a description of two new fuzz targets for configuration parser and generic card operations.

Part of the thesis was also an effort to integrate the implemented improvements into OpenSC upstream. The integration was successful for fuzz targets concerning parsing of the configuration file, PKCS #15 related functions, encoding of PKCS #15 objects, general card functions and OpenSC tools. The results of their run were evaluated based on internal OSS-Fuzz reports. A description of the corresponding pull requests to the OpenSC upstream can be found in Appendix C.

In conclusion, the goal of this thesis was achieved. A total of 7 fuzz targets were integrated into OpenSC. The fuzz target for PKCS #11 API

(Section 5.3) is part of the pull request to OpenSC upstream. These implementations have already contributed to the discovery of several bugs.

Further development of fuzzing support in OpenSC may include the integration of fuzz targets for specific card types (Section 5.4). It is also possible to replace the fuzz target `fuzz_pkcs15init` with the new version described in Section 5.2.4. For the tool `pkcs11-tool`, a fuzz target solution using SoftHSM is proposed in Appendix B.5. Its inclusion is feasible only after reworking `pkcs11-tool`, which should not contain terminations with `exit()` function.

Bibliography

1. KATZ, Jonathan; LINDELL, Yehuda. *Introduction to Modern Cryptography*. Chapman and Hall/CRC, 2021. ISBN 978-0-8153-5436-9.
2. SUSTEK, Laurent. *Encyclopedia of Cryptography and Security*. Hardware Security Module. Ed. by TILBORG, Henk C. A. van; JAJODIA, Sushil. Boston, MA: Springer US, 2011. ISBN 978-1-4419-5906-5. Available from DOI: 10.1007/978-1-4419-5906-5_509.
3. RANKLE, Wolfgang; EFFING, Wolfgang. *Smart Card Handbook*. 4th ed. The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom: John Wiley & Sons Ltd, 2010. ISBN 978-0-470-74367-6.
4. *OpenSC: Open Source Smart Card Tools and Middleware* [online] [visited on 2022-05-17]. Available from: https://github.com/OpenSC/OpenSC.
5. *OpenSC Wiki: Home* [online] [visited on 2022-05-17]. Available from: https://github.com/OpenSC/OpenSC/wiki.
6. TAKANEN, Ari; DEMOTT, Jared; MILLER, Charlie. *Fuzzing for Software Security Testing and Quality Assurance*. Norwood: Artech House, Inc., 2008. ISBN 978-1-59693-214-2.
7. *OSS-Fuzz: Continuous Fuzzing for Open Source Software* [online] [visited on 2022-05-15]. Available from: https://github.com/google/oss-fuzz.
8. *OpenSC Wiki: Overview* [online] [visited on 2022-05-17]. Available from: https://github.com/OpenSC/OpenSC/wiki/Overview.
9. WANG, Yongge. *Public Key Cryptography Standards: PKCS*. arXiv, 2012. Available from DOI: 10.48550/ARXIV.1207.5446.
10. OASIS. *PKCS #11 Cryptographic Token Interface Base Specification Version 3.0* [online]. 2020 [visited on 2022-05-09]. Available from: https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/os/pkcs11-base-v3.0-os.html.

11. *OpenSC Manual Pages: Section 1* [online] [visited on 2022-05-17]. Available from: http://htmlpreview.github.io/?https://github.com/OpenSC/OpenSC/blob/master/doc/tools/tools.html.

12. *OSS-Fuzz: Continuous Integration* [online] [visited on 2022-05-17]. Available from: https://google.github.io/oss-fuzz/getting-started/continuous-integration/.

13. *ClusterFuzz: Glossary* [online] [visited on 2022-05-17]. Available from: https://google.github.io/clusterfuzz/reference/glossary/.

14. *libFuzzer – a Library for Coverage-Guided Fuzz Testing. — LLVM 15.0.0git documentation* [online] [visited on 2022-04-30]. Available from: https://llvm.org/docs/LibFuzzer.html.

15. MCNALLY, Richard; YIU, Ken; GROVE, Duncan; GERHARDY, Damien. *Fuzzing: the State of the Art*. 2012. Tech. rep. Defence Science and Technology Organisation Edinburgh (Australia). Available also from: https://apps.dtic.mil/sti/citations/ADA558209.

16. CLARKE, Toby. Fuzzing for Software Vulnerability Discovery. *Department of Mathematic, Royal Holloway, University of London, Tech. Rep. RHUL-MA-2009-4*. 2009. Available also from: https://repository.royalholloway.ac.uk/items/4941b5d6-2f4a-8499-8954-1a7feee7cc4c/1/.

17. *tlsfuzzer: SSL and TLS Protocol Test Suite and Fuzzer* [online] [visited on 2022-05-15]. Available from: https://github.com/tlsfuzzer/tlsfuzzer.

18. LI, Jun; ZHAO, Bodong; ZHANG, Chao. Fuzzing: a Survey. *Cybersecurity*. 2018, vol. 1, no. 1, pp. 1–13.

19. *Efficient Fuzzing Guide* [online] [visited on 2022-04-30]. Available from: https://chromium.googlesource.com/chromium/src/testing/libfuzzer/+/HEAD/efficient_fuzzing.md.

20. THE CLANG TEAM. *Source-based Code Coverage — Clang 15.0.0git documentation* [online] [visited on 2022-05-15]. Available from: https://clang.llvm.org/docs/SourceBasedCodeCoverage.html.

21. DEMOTT, Jared. The Evolving Art of Fuzzing. *Def Con* [online]. 2006, vol. 14 [visited on 2022-04-30]. Available from: https://www.ida.liu.se/~TDDC90/literature/papers/evolvingfuzzing.pdf.

22. AIZATSKY, Mike; SEREBRYANY, Kostya; CHANG, Oliver; ARYA, Abhishek; WHITTAKER, Meredith. Announcing OSS-Fuzz: Continuous fuzzing for open source software. *Google Testing Blog*. 2016.

23. *The AFL++ Fuzzing Framework* [online] [visited on 2022-04-30]. Available from: https://aflplus.plus.

24. *AFL: American Fuzzy Lop - a Security-Oriented Fuzzer* [online] [visited on 2022-04-30]. Available from: https://github.com/google/AFL.

25. *honggfuzz: Security Oriented Software Fuzzer* [online] [visited on 2022-04-30]. Available from: https://github.com/google/honggfuzz.

26. THE CLANG TEAM. *Clang Compiler User's Manual — Clang 15.0.0git documentation* [online] [visited on 2022-05-15]. Available from: https://clang.llvm.org/docs/UsersManual.html.

27. *OSS-Fuzz* [online] [visited on 2022-04-30]. Available from: https://google.github.io/oss-fuzz/.

28. *OSS-Fuzz: Architecture* [online] [visited on 2022-05-15]. Available from: https://google.github.io/oss-fuzz/architecture/.

29. *ClusterFuzz* [online] [visited on 2022-04-30]. Available from: https://google.github.io/clusterfuzz/.

30. *Chrome Infrastructure - Monorail Issue Tracker* [online] [visited on 2022-04-30]. Available from: https://opensource.google/projects/monorail.

31. *ClusterFuzz: Architecture* [online] [visited on 2022-05-15]. Available from: https://google.github.io/clusterfuzz/architecture/.

32. *OSS-Fuzz: Setting Up a New Project* [online] [visited on 2022-05-07]. Available from: https://google.github.io/oss-fuzz/getting-started/new-project-guide/.

33. *ClusterFuzz: UI Overview* [online] [visited on 2022-05-15]. Available from: https://google.github.io/clusterfuzz/using-clusterfuzz/ui-overview/.

34. *OSS-Fuzz: ClusterFuzz* [online] [visited on 2022-05-15]. Available from: https://google.github.io/oss-fuzz/further-reading/clusterfuzz/.

35. *OSS-Fuzz* [online] [visited on 2022-04-30]. Available from: https://oss-fuzz.com/. [Internal for project members].

36. THE CLANG TEAM. *SanitizerCoverage — Clang 15.0.0git documentation* [online] [visited on 2022-05-15]. Available from: https://clang.llvm.org/docs/SanitizerCoverage.html.

37. *OSS-Fuzz: Bug Disclosure Guidelines* [online] [visited on 2022-05-15]. Available from: https://google.github.io/oss-fuzz/getting-started/bug-disclosure-guidelines/.

38. *GitHub Actions* [online] [visited on 2022-05-16]. Available from: https://github.com/features/actions.

39. FIORALDI, Andrea; MAIER, Dominik; EISSFELDT, Heiko; HEUSE, Marc. *AFL++: Combining Incremental Steps of Fuzzing Research* [online]. 2020 [visited on 2022-04-30]. Available from: https://aflplus.plus//papers/aflpp-woot2020.pdf.

40. *ClusterFuzz: libFuzzer and AFL++* [online] [visited on 2022-05-16]. Available from: https://google.github.io/clusterfuzz/setting-up-fuzzing/libfuzzer-and-afl/.

41. THE CLANG TEAM. *AddressSanitizer — Clang 15.0.0git documentation* [online] [visited on 2022-05-15]. Available from: https://clang.llvm.org/docs/AddressSanitizer.html.

42. THE CLANG TEAM. *UndefinedBehaviorSanitizer — Clang 15.0.0git documentation* [online] [visited on 2022-05-15]. Available from: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

43. THE CLANG TEAM. *ThreadSanitizer — Clang 15.0.0git documentation* [online] [visited on 2022-05-15]. Available from: https://clang.llvm.org/docs/ThreadSanitizer.html.

44. THE CLANG TEAM. *MemorySanitizer — Clang 15.0.0git documentation* [online] [visited on 2022-05-15]. Available from: https://clang.llvm.org/docs/MemorySanitizer.html.

45. THE CLANG TEAM. *LeakSanitizer — Clang 15.0.0git documentation* [online] [visited on 2022-05-15]. Available from: https://clang.llvm.org/docs/LeakSanitizer.html.

46. LARMOUTH, J. *ASN.1 Complete*. Elsevier Science, 2000. ITPro collection. ISBN 9780122334351.

47. JOHNSON, Don; MENEZES, Alfred; VANSTONE, Scott. The Elliptic Curve Digital Signature Algorithm (ECDSA). *International journal of information security*. 2001, vol. 1, no. 1, pp. 36–63.

48. *OSS-Fuzz Coverage Report 2021-11-03* [online] [visited on 2022-05-07]. Available from: https://storage.googleapis.com/oss-fuzz-coverage/opensc/reports/20211103/linux/src/opensc/src/report.html.

49. *OSS-Fuzz Coverage Report 2020-07-09* [online] [visited on 2022-05-07]. Available from: https://storage.googleapis.com/oss-fuzz-coverage/opensc/reports/20200709/linux/src/report.html.

50. *OSS-Fuzz Coverage Report 2020-07-09* [online] [visited on 2022-05-07]. Available from: %5Ctextbf%7Bhttps://storage.googleapis.com/oss-fuzz-coverage/opensc/reports/20200709/linux/src/report.html%7D.

51. *OSS-Fuzz: Code Coverage* [online] [visited on 2022-05-07]. Available from: https://google.github.io/oss-fuzz/advanced-topics/code-coverage/.

52. *OpenDNSSEC — SoftHSM* [online] [visited on 2022-05-09]. Available from: https://www.opendnssec.org/softhsm/.

53. *getopt_long(3): Parse options - Linux man page* [online] [visited on 2022-05-16]. Available from: https://linux.die.net/man/3/getopt_long.

54. *OSS-Fuzz Coverage Report 2022-05-16* [online] [visited on 2022-05-16]. Available from: https://storage.googleapis.com/oss-fuzz-coverage/opensc/reports/20220516/linux/report.html.

A Smart Cards

A smart card is a physical card with an integrated circuit (so-called chip). The cards can be divided into three categories based on their interface: contact, contactless, and hybrid. It is possible to classify the cards by their capabilities into memory and microprocessor cards [3]. Smart card technology is in detail described by ISO/IEC 7816 standard[1].

A.1 Storage

Access to the stored data is only possible via a defined interface. Sensitive data cannot be read from outside the card. They can be only used for specific operations directly on the card. Memory cards usually store data in EEPROM (Electrically Erasable Programmable Read-Only Memory). [3]

A microprocessor card has three types of memory – RAM (Random Access Memory), ROM (Read-Only Memory) and EEPROM. A logical directory and folder structure are available to access the data on the card. The root directory containing all files and directories is called the master file directory (MF). Other directories on the card are called dedicated files (DF). Elementary File (EF) stores the actual user data. Access control to the objects on the card is given by restricting access to individual files. [3]

A.2 Communication

Communication is always initiated by the terminal that sends commands to the card. The card only responds to these commands. The first message sent by card is ATR (Answer To Reset), initiated by the reset command from the terminal. Based on the ATR, the terminal evaluates the card information and parameters. [3]

The communication protocol between the reader and the card is called the APDU (Application Protocol Data Unit). The structure of the APDU is defined by the ISO/IEC 7816-4 standard. APDU is

1. Further information can be found at `https://www.iso.org/standards.html`.

divided into commands sent by the reader and responses sent by the card. The command contains a header and data. The response consists of data and returns code bytes. [3]

The PC/SC (Personal Computer/Smart Card) specification defines an interface for utilising smart cards in computer environments independently on platforms [3]. As an alternative to PC/SC, OpenSC also supports CT-API (Card Terminal-Application Programming Interface) [3, 4].

A.3 Application Areas

The usage of smart cards is widespread. The primary areas are payment systems, where the smart cards are used to perform transactions or as electronic purses. Another sector is telecommunications. Smart cards are also used in the GSM system known as SIM cards. In addition, smart cards can be part of ID cards and health insurance cards or can serve as a repository of cryptographic keys for authentication or signing [3].

A.4 PKCS Standards

PKCS stands for Public-Key Cryptography Standards, created by RSA Laboratories [9]. The standards are used for cryptographic implementations; they describe algorithms, protocols, data storage formats, APIs and work with tokens [9]. OpenSC supports smart cards compatible with PKCS #11 and PKCS #15 standards.

A.4.1 PKCS #11

"Cryptographic Token Interface Standard" [9] is a specification of API for cryptographic tokens. Cryptographic Token Interface is also known as Cryptoki. It is designed for devices storing cryptographic data and performing cryptographic operations. Cryptographic devices can perform cryptographic operations following the commands passed through the device driver. Cryptoki unifies the interface to work with the cryptographic device for application – it hides the implementation

details of the underlying driver layer. The interface provides a uniform environment and ensures portability. [9]

Cryptoki defines data types, logical objects, and functions. Objects include data, keys, and certificates. The API contains functions to work with these objects. The standard then describes the API and ABI for cryptographic and token operations such as signing, encryption, key and PIN handling. The non-public objects are accessible with a PIN (there are two possibilities – Security Officer PIN and standard user PIN). Some objects (as private keys) are not accessible in plaintext at all. [9]

OpenSC provides PKCS #11 module that implements the specified API. It is utilized in OpenSC `pkcs11-tool`, which is used for managing PKCS #11 compatible tokens. The `pkcs11-tool` can also load foreign PKCS #11 module. [11]

A.4.2 PKCS #15

"Cryptographic Token Information Syntax Standard" [9] describes how the application works with the token. It defines file structures that enable compliant card work unified [9]. It can be implemented on Integrated Circuit Cards defined in ISO/IEC 7816-4 standard.

The file structure on the card compatible with PKCS #15 has a root directory called the master file directory (MF). It must contain an Object Directory File (ODF) with pointers to other files, which serve as directories for files storing references to objects – Private Key Directory Files (PrKDFs), Public Key Directory Files (PuKDFs), Secret Key Directory Files (SKDFs), Certificate Directory Files (CDFs), Authentication Object Directory Files (AODSs) and Data Object Directory Files (DODSs). The structure must also have files storing information about the token and hold optional elementary files to track unused space in existing files. [9]

OpenSC implements several tools for working with PKCS #15 compatible cards. These are `pkcs15-tool`, `pkcs15-crypt` and `pkcs15-init`. With `pkcs15-init`, users can initialise PKCS #15 compatible cards if OpenSC supports them, e.g. there is `pkcs15-init` driver for the given card. For cards not using PKCS #15 format, OpenSC implements an emulator for this layer. [4]

B Other Implemented Improvements

This chapter contains the rest of implemented improvements for OpenSC fuzzing support beyond the scope of the mandatory part of the thesis.

B.1 Improvement of the Virtual Reader

The virtual reader parses fuzzing input as APDU responses from the card. There were only four basic operations that virtual reader implements – `connect`, `transmit`, `disconnect` and `release`. Those operations are crucial to reader work. [4]

Other operations that the reader can perform are `lock` and `unlock`. These operations are used in the `sc_lock()` and `sc_unlock()` functions during some of the procedures in OpenSC. Once the lock is obtained, the `card_reader_lock_obtained()` function is called [4]. Since the virtual reader does not support lock and unlock operation, fuzzing cannot reach the corresponding procedures in card drivers in fuzz testing [48].

To fix that and increase the code coverage in card drivers, it is needed to implement a lock and unlock operation in the virtual reader. The newly added `lock` and `unlock` use the same approach as in the reader driver for CT-API – the operations only return `SC_SUCCESS` value. The functions `sc_lock()` and `sc_unlock()` interpret the return value as the lock is obtained successfully and enable the usage of corresponding card functions.

Run Analysis After integrating this change into OpenSC, the corresponding functions called from `sc_lock()` are now covered in some card drivers such as `card_piv.c` according to OSS-Fuzz [54].

B.2 Improvement of `fuzz_pkcs15_decode` Fuzz Target

Existing fuzz target `fuzz_pkcs15_decode` aims to test the decoding functions for PKCS # 15 objects stored on a card [4]. However, according to statistics presented in Chapter 4, it has low code coverage.

According to the code coverage report from OSS-Fuzz [48], functions reachable from the fuzz target are not fully covered. A possible explanation is that the used internal representation of the PKCS #15 structure on the card is not fully initialised. Its type is `struct sc_pkcs15_card`, and besides the card information, it also holds objects representing file structure from PKCS #15 standard. The original PKCS #15 card object initialisation only allocated memory and set the required pointer to the general card object. The object, therefore, remains almost empty, and further card information required by some code paths is missing.

Also, the fuzz target contains the decoding of a public key object that requires an allocated public key structure. However, the object remained filled with zeros after the structure was allocated. This means that the type of algorithm is also set to the same value (RSA as the used macro has a value of 0); therefore, the decoding function never tests other algorithm types.

Structure of Fuzz Target `fuzz_pkcs15_decode` The first change to the fuzz target is separating the input into two parts, the buffer and the APDU responses. Its length is determined by the first two bytes of the fuzzing input. This data buffer is then used as an input for testing decoding functions.

It is necessary to simulate the card's connection to completely initialise the card object representing the PKCS #15 card structure, since the card information is obtained during the initialisation process. The virtual reader from `fuzz_pkcs15_reader` is created with the rest of the fuzzing input. The initialisation of PKCS #15 card object is then processed in `sc_pkcs15_bind()` function. After that change, the PKCS #15 card object contains card data, which can be used in tested functions.

When testing the decoding of a public key object from the data buffer, it is possible to do so in an iteration over available algorithms. Then, the public key structure is set with different algorithm types in every cycle. Decoding functions that use a `switch` statement for corresponding algorithm types should fall into algorithm-specific functions.

Corpus Initially, there was no corpus to the fuzz target. As the target begins with extracting data buffer for decoding functions, the corresponding encoding functions can output data from objects stored on the card to create these buffers. The resulting bytes were used to create a buffer part of the corpus.

The APDU parts of new `fuzz_pkcs15_decode` corpus files are created using previously generated APDU traces, which are part of the `fuzz_pkcs15_reader` corpus. Only the part corresponding to the card connection was used for our purposes. The exact number of bytes to truncate from the original files was calculated using the `fuzz_pkcs15_reader` debug logs.

Run Analysis Although this change causes a decrease in the speed of fuzz target testing to 500 executions per second [35], there is the desired coverage of functions that decode the public key object for other algorithms [54]. Furthermore, fuzzing also reaches parts of the decoding functions that are only executable when the PKCS #15 card object is initialised correctly [54]. The region coverage increased to 16% on average [35].

B.3 Configuration Parser

Configuration is crucial for the proper operation of OpenSC. Particular tools have a direct setup within command-line options and environment variables (`OPENSC_CONF`). The main setup for OpenSC is through a configuration file, which is maintained during compiling and installation. Users can change the location of `opensc.conf` file via environment variables or in a phase of preparation for compilation within the `./configure` command (`--sysconfdir` option with specification of path) [4].

The configuration file itself has a specific format – scconf system. It has a block structure, where the block's content is one of the following items – comment, list or another block [4].

Parsing of the configuration file is not tested by any fuzz target according to OSS-Fuzz coverage reports [48].

B.3.1 Structure of Fuzz Target for Parser

Parsing in OpenSC is done by two functions while establishing an OpenSC context. The `scconf_parse()` function works directly with the configuration file. The `scconf_parse_string()` uses the same parsing engine but avoids direct work with the file. For simplicity and better performance fuzz target calls `scconf_parse_string()`.

Since `scconf_parse_string()` takes a simple string as an argument, the target adjusts the buffer with zero character as the ending byte.

Configuration files containing all essential settings reach a size of about four kB. For testing, inputs with large sizes in megabytes lose their meaning due to arduous debugging of the test cases from found crashes. The limit for file size was set to 16 kB after discussion with project maintainers. They also suggested the usage of dynamically allocated memory rather than static buffer[1].

Corpus The initial corpus contains two basic and correct OpenSC configurations stored as text. They differ in length to cover various options. Both are based on the template from OpenSC.

B.3.2 Run Analysis

Before setting the input size, the average execution achieves around 700 executions per second. Fuzzing time is 53% probably due to 3 found crashes, which slow the run when frequently repeated [35].

After the size limitation, region coverage remains approximately the same. Execution speed increased above 2 000 executions per second and 99% fuzzing time [35].

B.3.3 Bugs Discovered by `fuzz_scconf_parse_string`

Fuzzing with `fuzz_scconf_parse_string` has already found several bugs.

1. Nevertheless, in general, dynamic memory usage slows the fuzzing procedure [14].

Local Run with libFuzzer Engine The local testing without corpus found a direct memory leak and out-of-bounds writes. The memory leak was connected to the incorrectly classified configuration data, which was subsequently not appropriately released.

OSS-Fuzz Run During the run in the OSS-Fuzz, a crash related to timeout was discovered. The problem is related to stack-overflow in too submerged recursion. The fix is to limit the immersion of the parser[2].

B.4 General Functions Performing Card Operations

The directory `libopensc/` consists mainly of particular card drivers. The drivers implement operations that the corresponding smart cards are capable of [4]. It is not convenient to access these provided functions and test them directly from fuzz target since preparing the environment for their calling is often necessary.

Therefore, OpenSC provides generic functions, that require card object `sc_card_t`, and allow carrying some generic and cryptographic operations. Some of those generic functions are covered by already existing fuzz targets, primarily with `fuzz_pkcs15_reader` [4]. Several functions appear not to be tested from OSS-Fuzz code coverage reports. They are also not reachable by the current fuzz targets [48].

Namely, the operations are:

- key wrapping and unwrapping,
- writing binary data into EF on the card,
- storing data on the card,
- listing file IDs from the DF into the buffer,
- getting a challenge from the card,
- and appending to file selected via a flag,

B.4.1 Structure of Fuzz Target `fuzz_card`

The mentioned functions require initialised general card object of type `sc_card_t`. Some of them also need a data buffer or an optional flag.

2. Details of fixed problem can be found in pull request `https://github.com/OpenSC/OpenSC/pull/2499`.

The fuzz target splits fuzzing input into three main parts – flag, challenge length and APDU responses. This is followed by the connection of a virtual reader working with the third part of the input. After setting the OpenSC context and the initialisation of the card object, the fuzz target calls `fuzz_get_chunk()` from the virtual reader API to obtain a data buffers for testing.

Then the tested functions are called one after the other. The operation for unwrapping keys require a data buffer, that is extracted via the `fuzz_get_chunk()` function. Another buffer is used also for testing `sc_put_data()`. The initially created flag specifies the file number to be written to.

Corpus Files from the original corpus for `fuzz_pkcs15_reader` involve APDU traces for the card connection, which can also be used in the corpus for `fuzz_card`. Unfortunately, APDU responses required for the particular tested functions can not be put together easily. The tested functions are mostly called internally in some OpenSC provided tools. It is not convenient to get the matching data from the debug logs. Therefore, corpus files contain additional bytes as malformed APDU responses after the part for the card's connection.

B.4.2 Run Analysis

The `fuzz_card` fuzz target 30 days after integration into the OpenSC project reports a speed of around 700 executions per second. The region coverage reaches 11.35%. The percentage of the regular crash is, on average, 80%. This is caused by crashes, which often recur. [35]

B.4.3 Bugs Discovered by `fuzz_card`

So far, fuzz target found a stack-buffer-overflow bug [35]. It is located in a function that handles ASN.1 tags in a list files operation.

B.5 Fuzzing `pkcs11-tool` with SoftHSM

The `pkcs11-tool` is one of the most used tools from the OpenSC project. Since the tool uses the PKCS #11 module (in the form of a shared library or linked statically), the card connection happens via

the `C_Initialize` function [4]. With `C_Initialize`, it is not easy to supply a virtual reader with fuzzed APDU card responses as it is by other fuzz targets for tools. The PKCS #11 API is covered by the fuzz target implemented in Section 5.3. So it is sufficient to test only the work of the tool itself. For this purpose, the software cryptographic module SoftHSM [52] can be utilised.

B.5.1 Structure of the Fuzz Target `fuzz_pkcs11_tool`

The SoftHSM communicates via PKCS #11 API [52], so it can be directly used by `pkcs11-tool`. The static SoftHSM library must be built, installed and linked directly to the fuzz target. Then, the user must create a configuration file for SoftHSM, set the destination folder for storing the token data, and use the `softhsm2-util` tool to initialise the token. In order to automatise this procedure, there is a basic bash script executing all the mentioned steps.

The fuzz target itself can directly fetch the provided API with `C_GetFunctionList` [10] function and use it in `pkcs11-tool`. From fuzzing input, the fuzz target only parses arguments for `main()` function.

Although the described fuzz target can test the `pkcs11-tool` source code, its direct integration into the OpenSC project is not possible. The `pkcs11-tool` contains many calls to the `exit()` function that is not compatible with fuzz testing, as the fuzzer crashes in those situations. Therefore, this fuzz target is only a proposal for future improvements to fuzzing support.

B.6 Standard Output of Tested API

There is no need to have the standard output stream open for fuzzing itself – it may cause an overfill of the user terminal during the local run and slow the immediate execution. On the other hand, the possible output can be convenient for users debugging the code, so it cannot be removed from the code entirely.

The optional way is to create an environment variable in the configuration phase of OpenSC when fuzzing is enabled. In that case, the fuzz target containing the following snippet closes the standard

output. However, it keeps it open when built in non-fuzzing mode, as shown in the following code snippet.

```
#ifdef FUZZING_ENABLED
    fclose(stdout);
#endif
```

In non-fuzzing mode, it stays open.

B.7 User Manual

Current fuzzing support in OpenSC does not contain any summary information on how it is possible to build OpenSC with fuzzing support and details on individual fuzz targets and their corpora. A manual in `README.md` file for users is created as part of this thesis. It describes how it is possible to build fuzz targets for fuzzing and local regression testing and generate new files into the corpus, including the possibility of generating own APDU traces and usable links to OSS-Fuzz libFuzzer documentation.

C Integration into OpenSC Upstream

The implementation part also includes the integration of some implemented fuzz targets into the OpenSC upstream. The integration takes place as part of pull requests to the upstream repository. This section contains a description of pull requests, discussions with the OpenSC team, possible changes in the source code, and necessary changes in the OpenSC project.

Pull Request for Config Parser Fuzz Target

- Pull request: `https://github.com/OpenSC/OpenSC/pull/2417`

Pull request is related to Section B.3. In contrast to the original design of the fuzz target, which contained a static buffer on the stack, the use of a dynamically allocated array was proposed and later used. Part of this pull request was also the inclusion of the `fuzzer.c` file and the setting of macro to limit the dump to standard output.

Pull Request for PKCS #15 Initialisation Fuzz Target

- Pull request: `https://github.com/OpenSC/OpenSC/pull/2500`

The fuzz target `fuzz_pkcs15init` is introduced in Section 5.2. After the pull request discussion, minor bugs such as checking return values were fixed. The rest of the comments concern fixes for locally found bugs.

Pull Request for Encoding and Card Operations Fuzz Target

- Pull request: `https://github.com/OpenSC/OpenSC/pull/2520`

This pull request contains code for `fuzz_pkcs15_encode` (Section 5.1), `fuzz_card` (Section B.4), and improvements for `fuzz_pkcs15_decode` (Section B.2). The main discussions and subsequent changes relate to when to extract data buffers from fuzzed input. The `fuzz_pkcs_decode` created a new buffer before each tested operation. The change involves moving the extraction buffer before connecting the virtual reader.

Pull Request for Tool's Fuzz Targets

- Pull request: `https://github.com/OpenSC/OpenSC/pull/2538`

The pull request integrates three fuzz targets testing some of the main OpenSC tools (Section 5.4). These fuzz targets are create for `piv-tool`, `pkcs15-tool` and `pkcs15-crypt`. The review concerns the mismatched `argc` variable and the clarification of bugs that fuzz targets found during the local run. There is also improvement regarding `sc_lock()` function from Section B.1.

Pull Request for PKCS #11 API Fuzz target

- Pull request: `https://github.com/OpenSC/OpenSC/pull/2550`

In addition to the new fuzz target presented in Section 5.3, the pull request fixes locally found issues.

D Data Attachments

The thesis archive has the following structure:

- the `targets/` directory with all created fuzz target source files,
- the `corpus/` directory holding corpora for particular fuzz targets,
- the `data/` directory containing APDU traces, data object files and log files,
- the `other/` directory with OSS-Fuzz patches and auxiliary scripts,
- and the `OpenSC/` repository with integrated fuzz targets.

Details of the compilation of OpenSC can be found in the `README.md` file, fuzz targets and corpora are further described in `targets.md` and `corpus.md`.