Fuzzing of PKCS#11 Trusted Application

KEVIN ZENG

MASTER´S THESIS

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY

FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Department of Electrical and Information Technology
Lund University, Faculty of Engineering

Master's Thesis

Fuzzing of PKCS#11 Trusted Application

Authors:
Kevin Zeng, yggdrasel@protonmail.com

Supervisors:
Christian Gehrmann, EIT, christian.gehrmann@eit.lth.se
Patrik Lantz, Axis Communications
Lars Persson, Axis Communications

Examiner:
Thomas Johansson, EIT, thomas.johansson@eit.lth.se

Time span:

Feb 2022 - Oct 2022

Abstract

The main goal of this thesis is to find an effective way to fuzz trusted applications (TAs) with source code residing in trusted execution environment (TEE). While fuzzing TAs has been previously done, no work has been found to utilize the source code of TAs to improve the fuzzing. Utilizing the source code in fuzzing can lead to an increase in code coverage compared to black-box fuzzing, and therefore could be more effective in testing critical parts of the software. This might inspire people to develop similar fuzzing techniques on TAs running on OP-TEE or other TEEs.

The fuzzing target will be the TA implementation of the Public-Key Cryptography Standard 11 (PKCS#11) currently developed by STMicroelectronics and Linaro [1]. The TA is complex, and no previous documents on any extensive security testing on the PKCS#11 TA has been found. The TA source code is also available to the public, which can be utilized by certain fuzzing techniques to empower the fuzzing process. The focus of the project will not solely be to fuzz the PKCS#11 TA specifically, but also a method to fuzz TAs in general.

The following list summarizes the goals of the project:

- Implement a proof of concept on how to fuzz a TA running on OP-TEE using fuzzing technique that takes advantage of available source code.
- Explore how to build an effective fuzzing harness which bridges the gap between the fuzzer and target expected input. The harness will also setup the necessary state of the target.

The solution provided in this thesis uses various external tools and projects to host, perform fuzz testing and provide insight on the target TA. The fuzzing process is able to explore deeper into the target and provide information related to bugs, code coverage and other fuzzing relevant information. However, in order to have a better fuzzing experience, certain highlighted problems of the project still needs attention. While the current state of the solution is not perfect, it is enough to serve as a proof of concept.

Popular Science Summary

Highly secure and sensitive tasks of applications in devices such as phones are often performed in a secure environment separated from normal tasks. This will provide additional protection making it harder for adversaries to manipulate or extract information. However, despite running in a protected environment, the applications are still vulnerable to attacks introduced by bugs. Therefore, this work will provide a method to security test these applications inside of such environments to increase the security making it more difficult for adversaries to abuse the system.

An application is set of instructions and data used to tell a computer to perform various tasks. However, can an application be trusted to perform the user-intended tasks? And in safe manner? For example, how can a user know that the password provided to a login prompt does not get forwarded to another person? Or if the password stored in the login authenticator can be leaked? To provide a higher trust- and protection-level, sensitive operations and data in normal applications can be moved to a trusted execution environment. This technique is commonly used in the mobile industry. Features such as mobile payments and fingerprint authentication offers higher security with the usage of trusted execution environments.

However, despite that the computer is executing the correct application in a protected environment, it might still be possible for an adversary to abuse. Gaps and flaws known as bugs can be introduced by the developers when creating the application, leaving security vulnerabilities that can be manipulated by an adversary to perform unintended or even harmful activities such as bypassing security features, leaking sensitive information and more.

This thesis focuses on finding bugs of applications designed for trusted execution environments in an automated way by using a concept called fuzzing. This is done by running the application in different test cases and monitoring its behaviours to find bugs. This can allow application vendors and security researchers to detect faults and patch them up to protect users from attacks. Certain industries might also require extensive security testing of application to provide a higher assurance in the products. The idea of the work is to inspire others to use similar methods to security test applications and

contribute to a safer digital world.

Acknowledgement

I would like to express my sincere gratitude to my supervisors and coworkers at Axis for providing me with help, guidance and discussions over the course of the thesis and made it a fun workplace. Especially my main supervisor Patrik who I have been working close with and has given me the opportunity to do research in this interesting area. I would also like to express my thanks to my supervisor Christian at the Department of Electrical and Information Technology at Lund University for a lot of valuable feedback on the thesis. Finally, I would like to thank my friends and loved ones for not only giving me feedback and help but also supporting me throughout this journey.

Abbreviation

Table 1: Abbreviations

Abbreviation	**Full Form**
AFL	American Fuzzy Lop
API	Application Programming Interface
ASAN	AddressSanitizer
CA	Client Application
CoT	Chain of Trust
DRM	Digital Rights Management
eFuse	Electronic Fuse
EL	Exception Level
ELF	Executable and Linkable Format
eMMC	Embedded MultiMediaCard
FEK	File Encryption Key
FS	File System
GCC	GNU Compiler Collection
GDB	GNU Debugger
GP	GlobalPlatform
HMAC	Hash-based Message Authentication Code
HSM	Hardware Security Modules
HUK	Hardware Unique Key
Intel SGX	Intel Software Guard Extensions
IPC	Inter-Process Communication
IV	Initialisation Vector
I/O	Input/Output
LSAN	LeakSanitizer
Mbed TLS	Mbed Transport Layer Security
MSAN	MemorySanitizer
Open-TEE	Open-Trusted Execution Environment
OP-TEE	Open Portable Trusted Execution Environment
OS	Operative System
OTP	One-time programmable

Continued on next page

Table 1 – continued from previous page

Abbreviation	Full Form
PID	Process Identifier
PKCS#11	Public-Key Cryptography Standard 11
POSIX	Portable Operating System Interface
PRNG	Pseudorandom Number Generator
PTA	Pseudo Trusted Application
QSEE	Qualcomm Secure Execution Environment
REE	Rich Execution Environment
ROM	Read-only Memory
RoT	Root of Trust
RPC	Remote Procedure Call
RPMB	Replay Protected Memory Block
SAT	Boolean Satisfiability
SCC	Security Critical Codes
SMT	Satisfiability Modulo Theories
SO	Security Officer
SoC	System-on-Chip
SSK	Secure Storage Key
syscall	system call
TA	Trusted Application
TC	Trusted Computing
TC	TrustedCore
TEE	Trusted Execution Environment
TF-A	Trusted Firmware-A
TLV	Type-length-value
TOCTOU	Time-Of-Check-Time-of-Use
TSK	Trusted Application Storage Key
TZ	TrustZone
uboot	Das U-Boot
UBSAN	UndefinedBehaviorSanitizer
UEFI	Unified Extensible Firmware Interface
UUID	Universally unique identifier

Introduction

Modern hardware is designed to be more compact, powerful and efficient, allowing developers to create more complex systems. Consequently, the increase in complexity entails a maintainability and updatability challenge resulting in a larger attack surface [2], thus a higher likelihood of discovering security vulnerabilities. Meanwhile, running *security critical codes (SCCs)* on devices (e.g. secure transactions and critical information handling) is more common [3]. These issues creates a demand for *Trusted Computing (TC)*, which is defined to help systems achieve secure computation, privacy and data protection [4]. The concept of *Trusted Execution Environment (TEE)* is later introduced as a way of addressing TC. It is an environment that allows for secure, isolated and tamper-resistant execution of arbitrary code.

Currently, there are many different implementations of TEE available, such as:

- Open Portable Trusted Execution Environment (OP-TEE) [1], currently handled by Linaro.
- Kinibi [2], developed by Trustonic.
- TrustedCore (TC) [3], developed by Huawei. It's used in Huawei mobile devices.
- TEEGRIS [4], developed by Samsung. It's used in Samsung mobile devices.
- Qualcomm Secure Execution Environment (QSEE) [5], developed by Qualcomm. It's used in Android devices.

[1]`https://optee.readthedocs.io/en/latest/general/about.html`
[2]`https://www.trustonic.com/technology/`
[3]`https://www.usenix.org/system/files/woot20-paper-busch.pdf`
[4]`https://developer.samsung.com/teegris/overview.html`
[5]`https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/guard_your_data_with_the_qualcomm_snapdragon_mobile_platform2.pdf`

- Intel Software Guard Extensions (Intel SGX) [6], developed by Intel.
- Apple Secure Enclave [7], developed by Apple.
- ZAYA Secure Operative System [8], developed by ZAYA, offers a TEE for RISC-V architecture.

A TEE can have many different use-cases, such as secure fingerprint functionality for mobile devices, digital rights management (DRM) technologies for protecting the media content of streaming services or secure storage for sensitive information in embedded and mobile devices.

Often, the security critical operations performed by most applications are done in an unsafe environment, also known as a *Rich Execution Environment (REE)* (e.g. Linux) that allows adversaries to perform modification or extraction of the sensitive information. For example, your password manager might be unlocked using a cryptographic object (e.g. a cryptographic key) which the adversary can steal from your running process. Therefore, there is a need to manage sensitive operations (e.g. related to key management) in a *Trusted Application (TA)* running on a TEE. An example of a TA implementation could be the PKCS#11 API [5] that handles the communication to cryptographic security tokens such as smart cards, USB keys, and hardware security modules (HSM), where the actual cryptographic operations are done. A PKCS#11 TA can be implemented for a specific TEE. For example, OP-TEE, a TEE designed as a companion to a non-secure Linux kernel running on Arm [6, p. 3]. OP-TEE uses TrustZone, hardware capabilities on newer Arm processors (e.g. Cortex-A) to create two separate execution environments which they call the "normal world" and the "secure world".

Many applications (e.g. Mozilla Firefox, OpenSSL and OpenVPN) uses the PKCS#11 to communicate with security tokens. In the context of OP-TEE, since the PKCS#11 TA can take input from the REE, any vulnerability in the TA itself could be a security threat to the massive user base of such applications. Therefore, it is important to test the PKCS#11 TA for bugs and vulnerabilities. An effective way to achieve this is by utilizing fuzzing

[6]`https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html`

[7]`https://support.apple.com/guide/security/secure-enclave-sec59b0b31ff/web`

[8]`https://riscv.org/news/2021/09/zaya-now-supports-risc-v-zaya/`

methods [7], [8], a dynamic testing method used for automating software tests and finding security vulnerabilities in software. In the context of security, the testing process involves providing a great number of inputs covering a large area of the application. The goal is to find edge cases that are not tested, and monitor unexpected events and metrics of the application. These events often expose implementation and design faults which could lead to security vulnerabilities that can be abused by adversaries.

This thesis is about finding a method to security test TAs efficiently using fuzzing techniques that utilizes the source code of such applications to empower the fuzzing process. This can allow different software vendors to security test TAs using similar methods.

Contents

Method

The method of achieving the goal of the thesis is presented below. The method will include the evaluation of different approaches, finding the relevant tools and integrating the environment and tools for a complete solution.

2.1 Fuzzing Environment Setup

An analysis on how to setup an environment that is able to host a TA and allowing it to be efficiently fuzzed will be performed. The design of the solution will focus on efficiency and simplicity. The design will provide the TA to be invoked by a client application (CA) and perform the intended operation. The environment will also have to support the usage of tools required by the fuzzing process.

2.2 Fuzzing Target

The fuzzing target (PKCS#11 TA) will have to be ported to the fuzzing environment solution. The porting will be focusing on the important and commonly used features of the target TA.

2.3 Fuzzing Tool Selection

When the environment for the TA and fuzzer is setup, modern source-code-utilizing fuzzing technology will be reviewed and used for the project. The selection is based on its performance and other properties (e.g. the nature of the fuzzing technique and availability etc.).

2.4 The Fuzzing Process

After the preparation of the environment and tools to use, a harness will be designed for the fuzzing target (PKCS#11 TA). This includes providing the fuzzer with information about the input structure and setting up PKCS#11 TA in states that can be efficiently fuzzed.

Background

To design a solution for the thesis problem, some insight of relevant concepts, tools and projects will allow the reader gain a better understanding of how some of the core technology works, the core problems and different possible solutions for them. In this section TrustZone and secure boot, two underlying technology used by OP-TEE to realize a TEE are introduced to allow the user to have a better understanding of the TEE concept. A specific implementation of secure boot, Trusted Firmware-A, which can be used with OP-TEE is shortly touched upon, to provide the reader with a concrete example. Since the target TA is designed to be ran in OP-TEE, some architectural information and tools in OP-TEE are presented to allow the user to better understand relations and features that a TA running in a TEE might be using or could be dependent on. The design of Open-TEE engine will be explained as this will be used as a part of the final solution. Since the target is a TA, the interface of such applications will be introduced. The target, PKCS#11 TA, will also be threat modeled to discuss possible attack scenarios on the target. As fuzzing is the chosen testing method for the target application, the concept of fuzzing will be introduced. Specific information about the fuzzer of choice for the solution will be presented together with tools that can be used with it to improve the fuzzing. Some information on the target, PKCS#11 TA, will also be introduced. Finally, some other relevant projects on the topic of fuzzing TAs will be mentioned.

3.1 TrustZone

TEEs such as OP-TEE utilizes the Arm hardware technology, TrustZone, to perform security checks on a hardware level in order to realize a TEE. The Arm core is abstracted into two *virtual cores (VCPUs)*, a secure VCPU and non-secure VCPU [4, p. 61]. Hence, a physical processor with TrustZone capabilities is able to create a separation of execution worlds: the normal world (in non-secure mode), where the REE (e.g. Linux) operative system (OS) resides, and the secure world in secure mode, where the TEE (e.g.

OP-TEE) resides [9]. This allows for an isolated safe execution environment on a separate kernel for authorized TAs.

3.1.1 Separation of Execution Worlds

Although the secure OS is executed alongside the non-secure OS, it is shielded from it and they are unable to run at the same time. Specifically, there is a master and slave relationship between the two worlds [10, p. 5]. The separation of environment is one of the core components of a TEE, and satisfies the *Separation Kernel Protection Profile (SKPP)* [4, p. 58]. The main requirements are:

“

- *Data (spatial) separation.* Data within one partition cannot be read or modified by other partitions;
- *Sanitization (temporal separation).* Shared resources cannot be used to leak information into other partitions;
- *Control of information flow.* Communication between partitions cannot occur unless explicitly permitted;
- *Fault isolation.* Security breach in one partition cannot spread to other partitions.

”

The Arm TrustZone extension adds a transitional mode, the monitor mode, to the existing Arm operation modes (see Table 3.1), which resides only in the secure world [11, p. 70, 12]. Its purpose is to provide an interface between the two worlds and manage transitions between the worlds, ensuring that the state of the world before a transition is properly saved and that the state of the world after the transition is properly restored [10, pp. 5–6]. As the *secure monitor* in monitor mode operates on the privileged instruction *Secure Monitor Call (SMC)*, any code execution in the secure world must be requested from an application in the normal world via a SMC and be permitted by the non-secure OS. This allows access to TAs without exposing secure resources (e.g. encryption keys) to the normal world. Figure 3.1 depicts a high level overview of the possible processor modes in the respective exception levels of an ARMv8-A architecture as well as an example of a normal world

Table 3.1: Processor Modes in ARMv8-A architecture

Processor Mode	Exception Level	Security State	Description
User (USR)	EL0	Both	The usual ARM program execution state, and is used for executing most application programs.
Supervisor (SVC)	EL1	Both	A protected mode for the OS.
System (SYS)	EL1	Both	A privileged user mode for the OS.
Abort (ABT)	EL1	Both	Entered after a data abort or prefetch abort.
Interrupt Request (IRQ)	EL1	Both	Used for general-purpose interrupt handling.
Fast Interrupt (FIQ)	EL1	Both	Used for handling fast interrupts.
Undefined (UND)	EL1	Both	Entered when an undefined instruction exception occurs.
Hypervisor (HYP)[1]	EL2	Non-secure only	Used by a hypervisor, that controls, and can switch between Guest Operating Systems that execute at EL1.
Monitor (MON)[2]	EL3	Secure only	A Secure mode for the TrustZone Secure Monitor code.

[1] Hypervisor is optional.
[2] Added with the TrustZone extension.

application making a call to an application in the secure world. Different ARM architecture can use different terminology and model for the TrustZone. As OP-TEE can be configured for ARMv7-A and ARMv8-A architecture, the ARMv8-A conventions will be followed for consistency. The table 3.1 will consist of the processor modes available in the ARMv8-A architecture.

3.1.2 Resource Access Control

Except for the different processor modes, there is also a *Non-secure bit (NS-bit)* set in the *Secure Configuration Register (SCR)*, which is unreachable from the normal world, to determine the security state of the processor execution

[13, p. 39]. An exception is the monitor mode, which the NS-bit has no effect on, since it always operate in the secure world. The physical address space in peripherals and memory are divided into a secure and non-secure, accessible only in the correct security state [10, p. 6]. This is enforced by the core by setting the NS-bit to 1 during a memory transaction created by the normal world. Whereas, secure world applications are able to access non-secure memory regions by setting the NS-bit and *NSTable-bit* in its *translation table entries*. Since secure and non-secure entries can co-exist, any attempt from the normal world to access secure data will be rejected and cached secure data will result in a cache miss.

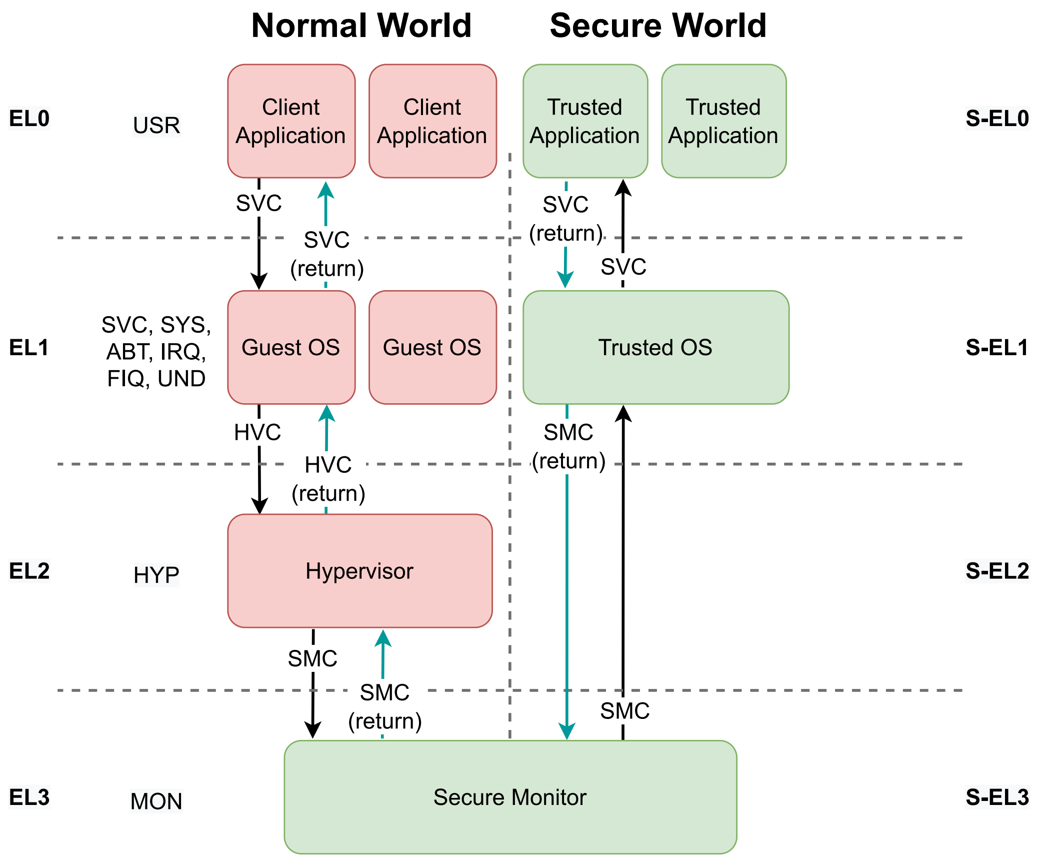

Figure 3.1: Overview of how the processor modes relate to the exception levels. An example of a normal world application invoking an application in the secure world is shown in the figure.
Red elements belong to the normal world. Green elements belong to the secure world.

3.2 Secure Boot

To provide protection from attacks being performed during a powered down state of the system (e.g. tampering with the secure world software image in flash memory) TrustZone is combined with a secure boot sequence to ensure platform integrity [13, p. 69]. The secure boot can validate each component of hardware and software in the normal and secure world using *Public-Key-*

Cryptography Standards (PKCS), generating a *chain of trust (CoT)* [13, p. 69, 14]. However, it is essential to establish a *Root of Trust (RoT)* at the beginning of the CoT, which is a trusted foundational security component that is difficult to tamper with [15, 13, p. 69]. The RoT integrity must hold, or else the CoT will be broken. The integrity control of the bootstrap process will start at the RoT and work its way up the CoT until the full initialization of the system has been performed. Figure 3.2 [13, pp. 5–5] shows a secure boot sequence of the secure world, ending the CoT when the secure OS is up and running. Each module is signed using some private key provided by a trusted vendor, and the signature is stored in the system [13, pp. 70–71]. At each new stage before execution, the binaries of the module will be integrity checked using the corresponding public key of the vendor. If any integrity control fails during the initialization process, the CoT breaks signifying that the binaries has been modified and thus differing from the vendor specified ones. As a result, the bootstrap process is interrupted.

In the OP-TEE project, the RoT is the *read-only memory (ROM) system-on-a-chip (SoC)* bootloader [9]. The ROM SoC will start the initialization of peripherals (e.g. memory controller). It will then boot the OP-TEE OS (in the secure world) stored in a flash memory. The secure OS will perform security checks before booting the Linux OS (in the normal world). This will ensure that the normal world applications does not perform any modifications prior to the integrity controls.

3.2.1 TF-A

The Trusted Firmware organisation provides reference implementations of the secure world software (running in S-EL3), for building the foundation to a TEE [16]. For example, *Trusted Firmware-A (TF-A)* a reference implementation for ARMv7-A and ARMv8-A architectures, which is used in Cortex-A and Neoverse processors. The TF-A provides an implementation of a secure boot process which can be used with a secure OS as OP-TEE. An overview of how the CoT is built starting from the RoT is illustrated in figure 3.3 [17, 18].

In TF-A, the *initial boot loader* in stage BL1, is stored in ROM and is the first code to be executed in the system [19, p. 47]. The initial boot loader will prepare to load and authenticate a *trusted boot firmware* image, of stage BL2, into RAM. This firmware can be updated allowing for vulnerability

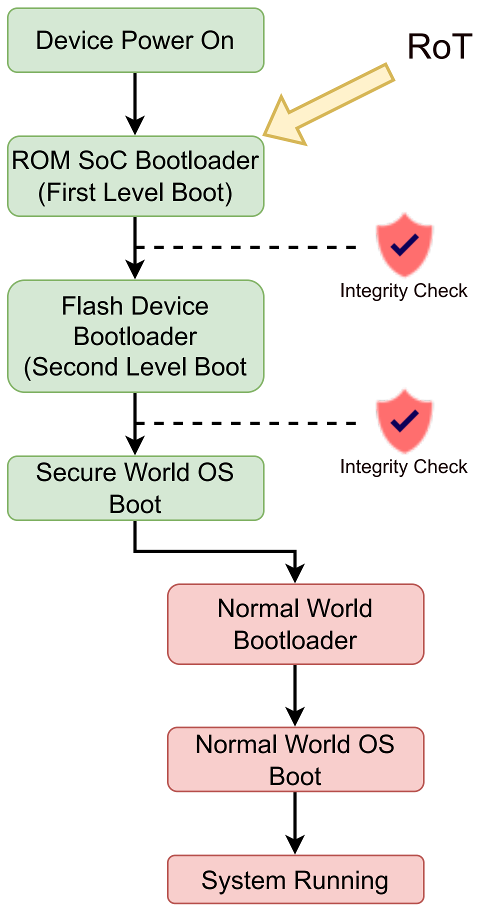

Figure 3.2: The figure shows a typical boot sequence of a TrustZone-enabled processor. The secure boot scheme adds cryptographic checks to each stage of the secure world boot process. Note that it is also possible to configure additional integrity checks in the normal world boot stages.

patches. The control of execution will be handed over to the trusted boot firmware.

The trusted boot firmware will prepare, authenticate and load all the EL3 firmware, of stage BL31, into RAM [19, p. 48]. This includes the EL3 secure monitor run-time firmware. Then the control of execution will be handed over to the secure monitor.

The secure monitor firmware will now load the S-EL1 payload of stage BL32, which is often a secure OS. Similarly to the previous stages, initialization preparation and authentication of the secure OS is performed. Next, the execution control is handed to the secure OS.

Finally, the secure OS will load a boot loader for the non-secure OS such as *Unified Extensible Firmware Interface (UEFI)* or *Das U-Boot (uboot)* in stage BL33.

3.3 OP-TEE

The OP-TEE project, is an open source project that started 2014, and currently maintained by Linaro[1] [20]. The Arm technology TrustZone is used by OP-TEE to make up a complete TEE. The OP-TEE OS operates in a confined TEE cooperating with a Linux OS in the REE.

To simplify TA development, the OP-TEE offers libraries and example TAs by default. In addition, several other tools (e.g. GDB), CAs (e.g. xtest) and TAs (e.g. PKCS#11) can be enabled in a configuration file for the build. The OP-TEE build can also be configured to include a TA Development Kit to simplify development and integration of TAs. The kit allows the generation of signed TAs from their respective source files with the included libraries, header files and makefile scripts [21]. The design of the OP-TEE adheres to the widely accepted industry standard set by the GlobalPlatform (GP) [6, p. 49]. It implements the GP API specifications[20]. "OP-TEE implements TEE Internal Core API v1.1.x which is the API exposed to TAs and the TEE Client API v1.0, which is the API describing how to communicate with a TEE" [20].

[1]Linaro's official website `https://www.linaro.org/`.

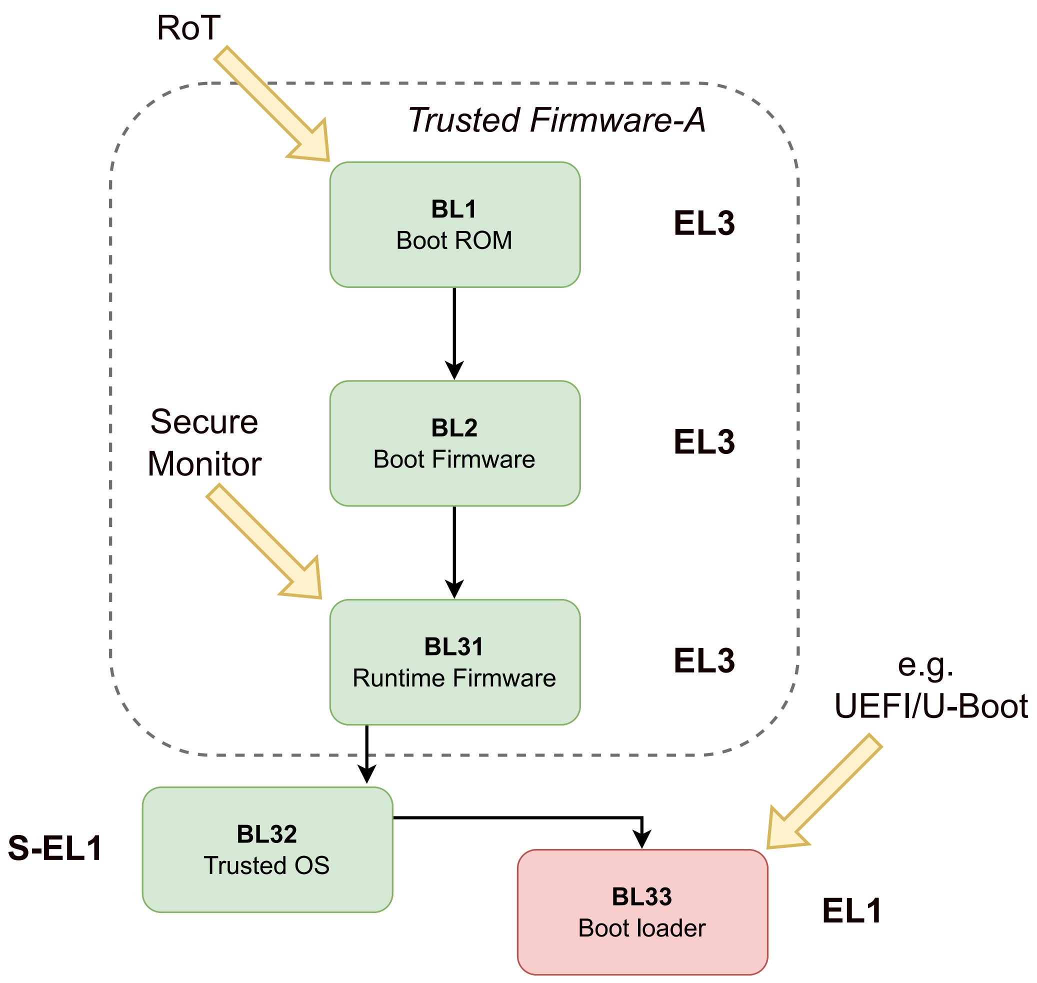

Figure 3.3: The figure illustrates the verification flow creating the chain of trust in Trusted Firmware-A starting from the root of trust.

3.3.1 Architecture

The major components of OP-TEE are illustrated in figure 3.4 [18].

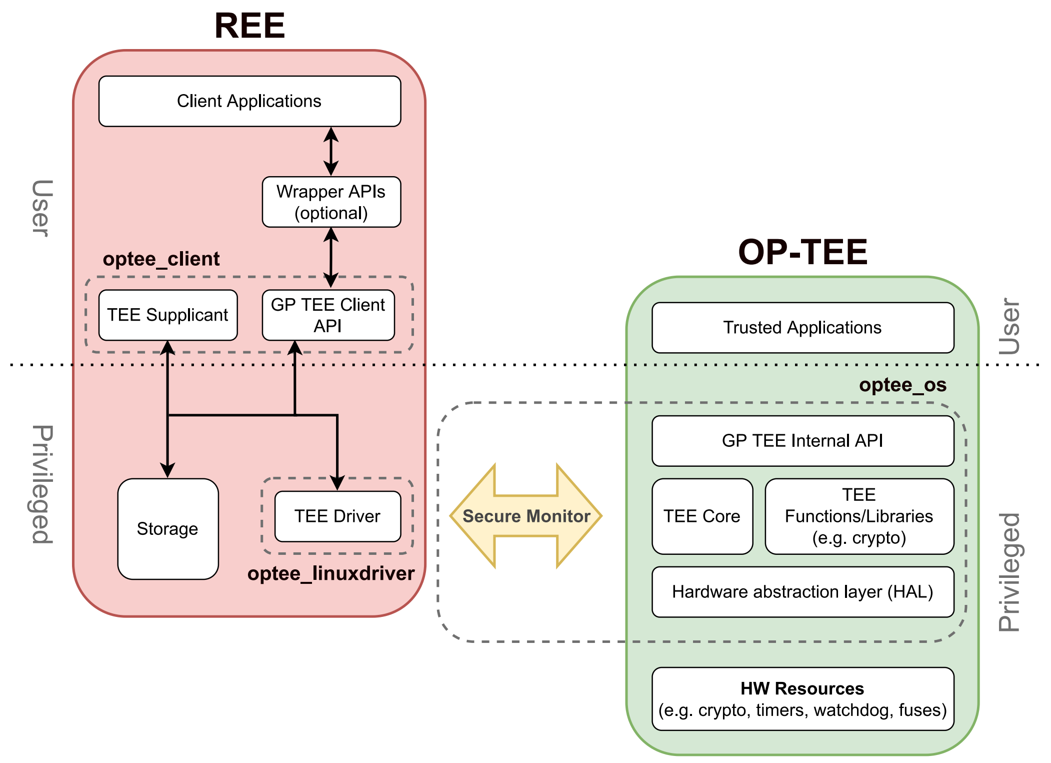

Figure 3.4: An overview of the OP-TEE architecture.

The main component of OP-TEE is the OP-TEE core which runs in secure kernel mode. Trusted utility libraries are provided by OP-TEE for TA development [21]. For example, the GP device TEE internal core API library that provides standard TA services which allows TAs to call secure services executing at a higher privilege. It implements the GP TEE internal API and is an interface for TA communication with the secure OS via system calls (syscalls). The loading of static and dynamic libraries are also supported.

OP-TEE also implements the GP device TEE client API specification for Linux based OS in form of a user space library and a Linux kernel TEE driver [21]. This allows TAs to be invoked via the API from CAs and a way to access OP-TEE core services from the normal world.

The TEE Linux device driver/subsystem (optee_linuxdriver) handles the details needed to communicate with OP-TEE [22]. The Linux subsystem handles registration of TEE drivers, manages shared memory between Linux and the TEE and provides a generic API to the TEE. However, it is difficult for OP-TEE, from the secure world, to perform certain actions in the normal world (e.g. accessing a non-volatile media device that is controlled in the normal world or accessing shared resources between the worlds). Therefore, a TEE supplicant is implemented in the normal world user space to handle remote services for the OP-TEE core. The TEE supplicant is invoked by the OP-TEE core via the TEE Linux kernel driver.

3.3.2 Trusted Storage

Objects created and handled by the PKCS#11 TA (e.g. tokens, keys, etc.) are stored and protected in a *secure storage*. These objects could be used for authentication of services or decryption of data and must be secured. OP-TEE has implemented its secure storage in accordance to the GP TEE Internal Core API, allowing storage of persistent general-purpose data (e.g. application-specific data) and key resources with confidentiality and integrity protection [23]. The secure storage also guarantees that all storage modifying operations are *atomic*, meaning that operations either completes in its entirety or no write operation will be done.

The secure storage, as a part of the GP trusted storage requirement (in the GP TEE Internal Core API), has to fulfill the following requirement:

“

1. The Trusted Storage may be backed by non-secure resources as long as suitable cryptographic protection is applied, which MUST be as strong as the means used to protect the TEE code and data itself.
2. The Trusted Storage MUST be bound to a particular device, which means that it MUST be accessible or modifiable only by authorized TAs running in the same TEE and on the same device as when the data was created.
3. Ability to hide sensitive key material from the TA itself.
4. Each TA has access to its own storage space that is shared among all the instances of that TA but separated from the other TAs.

5. The Trusted Storage must provide a minimum level of protection against rollback attacks. It is accepted that the actually physical storage may be in an insecure area and so is vulnerable to actions from outside of the TEE. Typically, an implementation may rely on the REE for that purpose (protection level 100) or on hardware assets controlled by the TEE (protection level 1000).

[6, p. 66] ”

3.3.2.1 REE FS and RPMB FS

There are currently two supported implementations of secure storage in OP-TEE: a REE file system (FS) in flash memory and a *replay protected memory block (RPMB)* FS in embedded *MultiMediaCard (eMMC)* device. Both modes can be used simultaneously.

An overview of the secure storage system architecture is illustrated in figure 3.5 [23]. All persistent objects stored in secure storage are integrity protected and encrypted [6, p. 67]. The process of storing a persistent object in the secure storage begins with a TA calling a write function provided by the GP Trusted Storage API. The write function in the GP internal API library will make a syscall, which is implemented in TEE trusted storage service [6, p. 66]. The TEE trusted storage service will in turn invoke a series of TEE file operations to store the data in the TEE FS. The TEE FS is an internal intermediate FS in the OP-TEE OS (S-EL1). It takes care of encrypting data and storing the the data in the normal world FS (REE FS or RPMB FS). The TEE FS will prepare the data to be sent to the normal world FS, by encrypting the data using the key manager and sending it using some (REE or RPMB) file operation to the TEE supplicant in EL-0 using a series of *Remote Procedure Call (RPC)* messages. A RPC message causes a subroutine to execute in a different address space. Upon receiving the RPC messages, the TEE supplicant will store the encrypted data in the normal world FS.

3.3.2.2 Storing TA in Secure Storage

TAs are stored in the normal world FS in Executable and Linkable Format (ELF) [6, p. 73]. They are identified by their universally unique identifier (UUID). The normal world FS is considered untrusted and a reason for storing TAs in normal world FS is due to the large size of TAs. In order to utilize

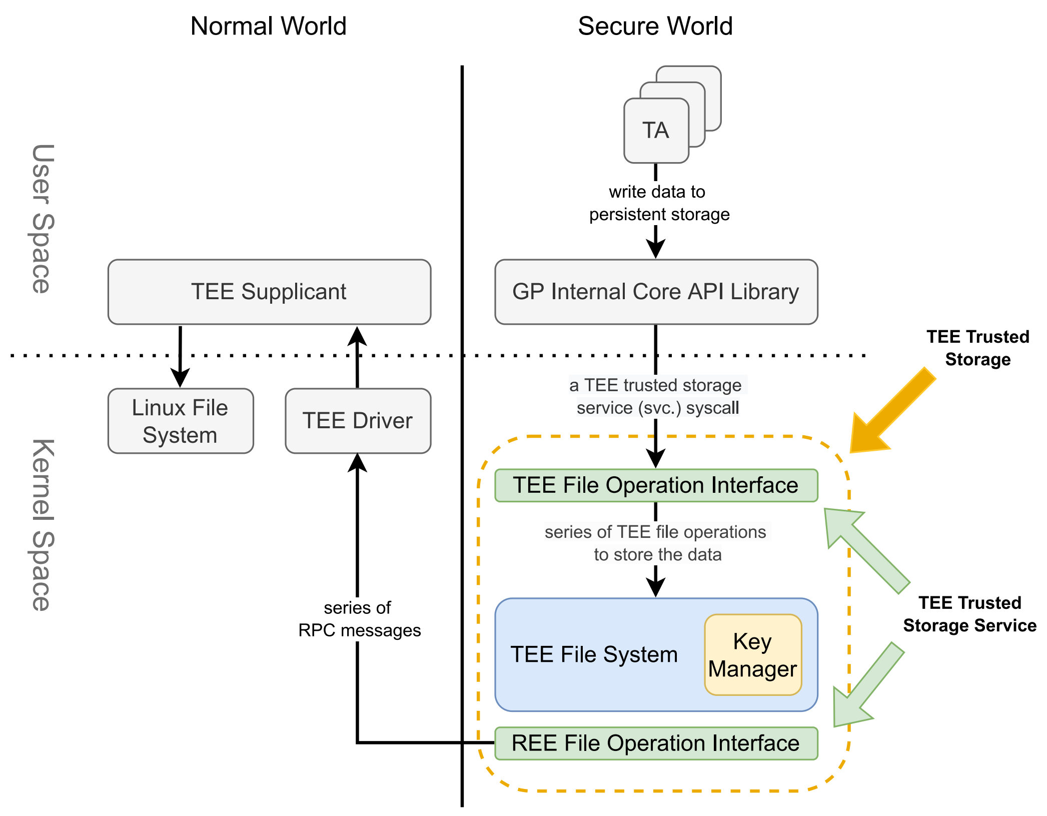
Normal World
Secure World
User Space
Kernel Space
TA
write data to persistent storage
TEE Supplicant
GP Internal Core API Library
TEE Trusted Storage
Linux File System
TEE Driver
a TEE trusted storage service (svc.) syscall
TEE File Operation Interface
series of TEE file operations to store the data
TEE Trusted Storage Service
series of RPC messages
TEE File System
Key Manager
REE File Operation Interface

Figure 3.5: The figure shows an overview of the secure storage system architecture. In this example the TA writes data to a persistent object which is later stored in the REE FS.

the normal world FS for TA storage the ELF files must be ensured to come from a verified vendor and not altered or corrupted in any way. Therefore, they are code signed and verified before being loaded into OP-TEE. The TAs can also be optionally encrypted for increased security [6, p. 73].

When the stored TAs in the REE FS are to be loaded into secure world, the OP-TEE core sends a series of RPC, via the OP-TEE driver, to the TEE supplicant, to allocate memory for a payload buffer for the TA [6, pp. 76–77]. The OP-TEE core then registers this payload as shared memory (into the TA address space) and commands the supplicant to copy the TA into a (non-secure) shared memory. The TA will then be loaded from shared memory into secure memory. Lastly, the OP-TEE core will make the supplicant free the payload it allocated.

Despite that TAs are signed or even encrypted an attacker is still able to load an outdated genuine TA with possible vulnerabilities to OP-TEE. RPMB protects against data rollbacks [4, p. 60]. A database file ta_ver.db stored in the OP-TEE core is used for TA version control. It will check for the TA version and prevent any downgraded TAs from being loaded. However, it will allow any newer versions of a TA with a version number higher than the one stored in the database. The upgraded version will replace the version number in the database.

3.4 The Open-TEE project

The Open-TEE project is a virtual TEE created by the *Secure Systems* research group at *Aalto University* in collaboration with the *Intel Collaborative Research Institute for Secure Computing*. It implements functionalities and components imposed by the GP. The implementation of these functionalities in Open-TEE might not always fulfill the requirement of GP, e.g. how resources are released during unexpected crashes of a TA [24, p. 5]. The actual TEE environment might utilize certain hardware of different characteristics [24, p. 6]. Hence, Open-TEE should not be used in production as a replacement for other TEE environments. Open-TEE guarantees that a working TA running on Open-TEE will also work on any GP-compliant hardware TEE [24, p. 2]. The virtual TEE requires Linux/GNU to run and has been tested on Arm and x86 architecture [25, 24, p. 6]. A TA developed by Trustsonic using Open-TEE has been compiled and used in production [26, p. 12].

3.4.1 Architecture

Open-TEE runs as a daemon process in the user space, abstracting away the TEE OS [24, p. 5]. When running a CA binary stored in the user space, Open-TEE will handle the invocation of TAs and the communication to them. The architecture of Open-TEE is illustrated in figure 3.6 [24, p. 5]. When initiating the Open-TEE engine, it will start a *base* which will then be forked into a *manager* and a *launcher* process. The manager will function as the secure world software, e.g. managing connections between CAs and TAs, monitoring the CA process status, provide a secure storage for a TA, handling shared memory regions for different sessions, and other functionalities imposed by the GP. The launcher process acts as prototype for new TAs. It allows a more optimized creation of TA processes, by pre-loading shared libraries (i.e. the Open-TEE shared library implementations of the GP internal core and client) and configuration of common components shared by all TAs in the launcher process. The TA will then be loaded, dynamically, as a shared library into the cloned launcher process, effectively becoming a TA process of its own. This process is then re-parented onto the manager process. This will allow the manager to take control of the TA allowing it to enforce the GP requirements. Each TA process will have two threads running: the I/O thread handling inter-process communication (IPC) and the logic thread, which is the TA working thread. The IPC used in Open-TEE are Unix domain sockets and inter-process signals to exchange information between the CA and TA and control the system.

3.5 TA

3.5.1 TEE Client and Internal Core API

TAs and CAs are unable to communicate directly because they execute in different environments. Instead, they follow the TEE client API and internal API for communication between the normal and secure world. Each TA must provide a few entry point functions, collectively called the TA interface. The entry points are called by the Trusted Core Framework. An extract of the client and internal core API can be found in table 3.2 and 3.3 respectively.

A CA will associate itself with a TEE in the secure world by creating a

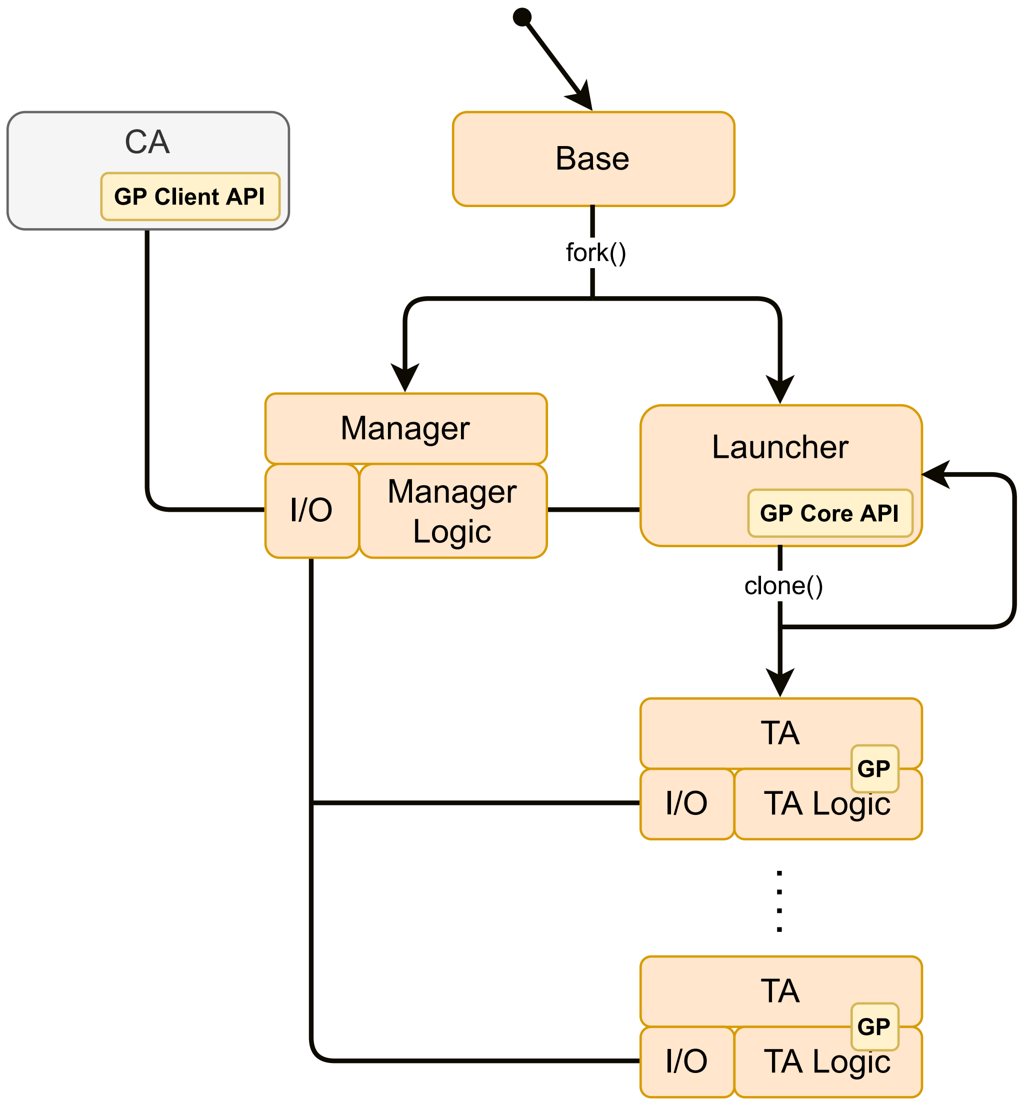

Figure 3.6: The figure shows an overview of the Open-TEE architecture. The orange components are part of the TEE OS while the yellow belong to the GP API.

logical connection called context. This is done by calling the function *TEEC_InitializeContext* in the CA [9, p. 3]. This will create an instance of the TA, which then calls *TA_CreateEntryPoint*. It is possible for a CA to initialize multiple contexts to different TEEs simultaneously.

Only after an initialized context can a session be established. A session is a logical connection between the CA and a specific TA and is created by the CA when providing a UUID to the *TEEC_OpenSession* function call. When a new session is created to a TA instance the *TA_OpenSessionEntryPoint* function call will be invoked. The UUID is used to differentiate different TAs since all TAs has unique UUIDs. A CA can establish multiple sessions to different TAs, however it is only able to have one session opened at a time. It is possible to have an initial data exchange with the TA and to specify connection methods (e.g. user authentication in the CA for increased access to data or functionality) [27, p. 11].

Within a session the CA is able to communicate with the TA using *shared memory* [9, p. 3]. Shared memory is a memory block shared between the normal and secure world and used to exchange data between CAs and TAs [6, p. 40]. The shared memory infrastructure is necessary because the secure and normal worlds have separate page tables. Hence, they cannot access common memory.

After a session is established the CA can communicate with the TA using the *TEEC_InvokeCommand* and *TA_InvokeCommandEntryPoint* functions [9, p. 3]. Shared memory is then used to send a numerical value to the TA, identifying a command to execute by the TA. The commands are mapped to defined function in the TA. The TA then performs the operation. An optional operation payload can be included by the CA, passed inside the operation parameters (which currently limited to 4) [27, p. 12]. When a command has been invoked, the CA thread is blocked waiting for results from the TA. However, several CA threads can be created to perform multiple commands for concurrency.

The operation parameters sent are either *memory references* or a *value parameters* with an associated data flow direction (e.g. input, output or bidirectional) [27, p. 12]. In case of memory reference, the direction will affect when the underlying memory buffers need to be synchronized with the TA. Shared memory buffers are used for data exchange using memory reference parameters. Meanwhile, value parameters simply exchanges two

32-bit integers without sharing nor synchronizing memory. Although, shared memory can be reused in multiple command invocations and sessions it is only possible as long the memory still exist within the scope of the TEE context [27, pp. 13–15]. When the called API function returns, then the bytes that the memory reference is referencing to is considered no longer alive, meaning that there is no synchronization and data corruption can occur.

The TA operation parameter received from a CA must be verified by the TA before usage [6, p. 144]. Expected parameters must be set in the TA. The type of the received parameters in the TA can be provided using the TEE_PARAM_TYPE_GET(param_types, param_index) macro. The index specified which parameter is checked. To verify that the TA receive correct parameters, a comparison with macros of the expected parameter can be compared against the actual received parameter. The verification include the flow direction of the parameter and its type. This is a security precaution, in case the CA is compromised and manipulated into sending unexpected data to the TA.

A memory reference can refer to a (pre-) registered memory, which is a region within a block of memory created before the operation, and a temporary memory reference, which is a CA owned memory portion that is immediately registered with the TEE Client API for the duration of the operation [27, p. 12]. It is nevertheless more efficient to use registered memory if the memory buffer will be used in more than one command invocation.

All session and invoke commands receive a return code indicating a successful operation or an error code of the fault [27, p. 12]. When the TA has finished performing the operation, the control is transferred back to the CA.

In order to end the communication between CA and TA properly the session should be closed by the CA and TA calling the *TEEC_CloseSession* and *TA_CloseSessionEntryPoint* functions. To finalize the context, the CA and TA calls the *TEEC_FinalizeContext* and *TA_DestroyEntryPoint* functions.

3.5.2 PTA

Pseudo TA (PTA) differ from normal TAs in that they are implemented directly in the OP-TEE core tree in core/pta. They are built into the OP-TEE core blob [6, p. 72]. PTAs run in S-EL1, thus with special privilege. They will have access to the same functions, memory and hardware as the

Table 3.2: An extract of GP client API functions

Function	**Description**
TEEC_InitializeContext	Initializes a new TEE Context, forming a logical connection between a TEE and CA.
TEEC_FinalizeContext	Finalizing the TEE context, releasing the logical connection stored in the context.
TEEC_OpenSession	Opens a new Session between the CA and the TA corresponding to the specified UUID.
TEEC_InvokeCommand	Invokes a command in the TA within the specified session using a command ID.
TEEC_CloseSession	Closes a Session by terminating the connection between the CA and TA stored in the session.

Table 3.3: An extract of GP TEE internal core API functions

Function	**Description**
TA_CreateEntryPoint	The constructor of the TA, creating a new instance of the TA. This is called once and only once in the lifetime of a TA instance.
TA_OpenSessionEntryPoint[1]	Called when a CA attempts to connect to a TA instance in the context and open a new session.
TA_InvokeCommandEntryPoint	Provides the service (using a command handler) requested by the CA during its command invocation using TEEC_InvokeCommand.
TA_CloseSessionEntryPoint	Called when the CA closes a session and disconnects from the TA instance.
TA_DestroyEntryPoint	The destructor of the TA. This is called by the Trusted Core Framework just before the TA instance is terminated.

[1] The open session request shall result in a new TA instance being created if the *gpd.ta.singleInstance* TA property is set to false. If set to true, a single TA instance is created for all client sessions.

OP-TEE core itself. PTAs are OP-TEE firmware services provided to the TAs running inside of OP-TEE. For example, the PKCS#11 TA utilizes a *pseudorandom number generator (PRNG)* PTA to generate additional seed material to the random number generator used in for example various cryptographic operations. The GP core internal API will not be available for PTAs, because the OP-TEE core is not linked to *libutee* which implements the TEE internal core API. Instead, they are strictly limited to the OP-TEE core internal APIs and routines.

3.5.3 Inter-TA Communication

Meanwhile it is not possible to spawn threads from a TA/PTA, it is possible in OP-TEE to run multiple TAs/PTAs in parallel. This allows them to communicate through the *TA2TA* interface [6, p. 165]. In this case a TA/PTA can act as a client to another TA/PTA. The *TEE_OpenTASession* is used to open a new session from a TA/PTA to another TA/PTA. The communication between TAs/PTAs uses the same process as the CA to communicate to a TA [28, p. 22]. In order to assure that the communication has not been exposed to the normal world to the receiving TA/PTA, an indicator is used. This allows the TA/PTA to trust the meta data associated with the received content (e.g. the caller TA UUID). Any TA/PTA in an external TEE will be considered a part of the REE, as there is no way to verify their trustworthiness. Similarly to the CA and TA communication commands are invoked by the *TEE_InvokeTACommand* and session has to be closed with a *TEE_CloseTASession*.

3.5.4 Threat Model

In this thesis the following threat model is assumed:

- The PKCS#11 TA running in OP-TEE is targeted. The adversary's goal is to extract information from the TA (e.g. from the secure storage).
- Anything running in the normal world is unsafe. This means the adversary may have user access, or even root access in the REE.
- The adversary is able to plant a malicious TA in the TEE..
- The system is physically protected and hence, physical attacks via debug interface or side-channel attacks is out of scope.

In order to protect a TA and its assets in the secure world security boundaries has to be enforced to protect the execution, memory, input/output (I/O) (e.g. peripherals), hardware (e.g. crypto engines).

1. The separation of REE and TEE. The REE must be prevented from compromising the TEE.
2. The separation of TA and TA. A malicious TA must be prevented from compromising other TAs in the secure world.
3. The separation of TA and TEE OS. A malicious TA must be prevented from escalating privileges. This can break the second separation.

In this thesis the implementation of OP-TEE will be excluded as a testing target, focusing only on the PKCS#11 TA implementation. The adversary is assumed to be targeting the secret of the PKCS#11 TA and is preparing an attack from a fully compromised REE. The adversary has access to the full source code of both the PKCS#11 TA and the OP-TEE project.

A possible attack scenario is to launch an attack from a CA since the adversary has control over the REE. A CA can be used to communicate with the target TA running in the TEE. If the TA has any security vulnerabilities, it can be abused by a malicious CA. A severe vulnerability can allow the attacker to extract sensitive data stored in for example the secure storage.

The PKCS#11 TA is also susceptible to an attack from another TA. The adversary might be able to plant a malicious TA in the secure world. In this case, the malicious TA might try to open a session directly to the PKCS#11 TA through the TA2TA interface, allowing it to test for vulnerabilities in the target TA.

In this thesis any indirect attack on the PKCS#11 TA, e.g. privilege escalation in OP-TEE or attack on the hardware platform its running on will be considered out of scope.

3.6 Fuzzing

There are many vulnerability discovering techniques that can be used on target application; hence, the application under test. Some of the more traditional techniques include: *static analysis*, *dynamic analysis*, *symbolic* *execution* and *fuzzing*.

When targets are analysed without execution it is considered to be static analysis [29, p. 2]. Analysis could be performed on features related to lexical, grammar, semantics. Data flow analysis and model checking are also considered static analysis. Tools can be used to perform the analysis. However, they are prone to false positives and false negatives, resulting in low accuracy. In contrast, dynamic analysis requires execution on target application by either running it in a real or an emulated environment. This method requires much domain knowledge and involvement from the analyst, resulting in low efficiency.

A symbolic execution symbolizes inputs to the target, allowing it to build a set of constraints for each execution path in the target. When the execution is done, constraint solvers such as *boolean satisfiability (SAT)* and *satisfiability modulo theories (SMT)* solvers will try to solve the constraints by finding inputs that can exercise the execution path. A common problem that arise in symbolic execution is the path explosion problem. It is the fact that the number of execution states increase exponentially to the scale of the target. Constraints in complex applications, especially with loops and recursions becomes hard to solve and time consuming. Instead, selective symbolic execution can be used on larger targets. Another limitation is that the lack of control of interactions outside the symbolic execution environment (e.g. syscalls, handling signals, etc.), leading to inconsistent outcomes. Hence, the main issue is scalability.

The currently more favored technique is fuzzing. A fuzzer starts by generating a large amount of inputs, including normal and abnormal ones, and feed them to the target application. The fuzzer will then monitor the execution states, trying to detect for exceptions or memory leak. Fuzzing techniques suffer from low efficiency and code coverage. However, it compromises by providing good scalability, accuracy and requires little knowledge of the inner workings of the target.

Fuzzing techniques are widely used in the security community. For instance, over 18,000 bugs in Google Chrome, over 11,000 bugs in the Linux kernel and over 1,800 bugs in Microsoft Office [30], have been discovered by various fuzzing tools. Windows regularly use white-box fuzzing techniques to find vulnerabilities for which they patch every month for their users.

3.6.1 The fuzzing process

The approaches and implementations of different fuzzers might variate greatly. However, the phases of a fuzzing process are the same and can be divided into 6 phases [31, pp. 27–28]:

1. identification of target
2. identification of inputs
3. generation of fuzzed data
4. execution of fuzzed data
5. monitoring of fault
6. determination of exploitability.

Identification of target. The first step, before deciding on a fuzzing tool or technique, is to choose a target. It could for example be an application, or a specific library or function within that application. Certain components might be shared across applications, indicating there is a bigger user base. If the target has not been previously extensively tested or if its vendor is known to leave many vulnerabilities in their products it could indicate possible vulnerabilities in the target, making it an interesting target.

Identification of inputs. Most application vulnerabilities are caused by user inputs [31, p. 28]. Therefore, locating the *input vector* (input locations) is vital; or else, the test case produced in the fuzzing process can be very limited. These inputs are potential fuzz variables.

Generation of fuzzed data. After identifying the input vector, test cases for the target need to be generated. This can take very different approaches and will be investigated in 3.6.2.

Execution of fuzzed data. When a test case is prepared, it is used to test the target by performing an execution (e.g. launching a target process with an input). In order to provide the target with test cases, an integration of the fuzzer and the target has to be manually performed. This is often done by bridging the two using a custom interface, often known as the *fuzzing harness*.

Monitoring of fault. Fuzzing a target with a fault monitoring process will allow the fuzzer to trace test cases that produced a bug (e.g. a crash, exception, etc.).

Determination of exploitability. From a security perspective, finding out what inputs can cause instability issues or bugs on the target is not enough. Triaging the bug in order to determine whether the bug can be exploited and turn into a security vulnerability is the utmost goal. This often requires domain knowledge. Certain tools (e.g. a debugger) can be used to assist the triaging. However, it usually requires much manual work.

3.6.2 Classification of Fuzzers

Fuzzing often falls into three categories: black-box, grey-box and white-box fuzzing [31, p. 3]. All three approaches require control of the input that goes into the target as well as the ability to observe resulting output. The first case treats the program as a black-box, lacking information about the target (e.g. the internal structure) [32]. The second case performs light-weight instrumentation on the target to gain information about the target. Meanwhile, the third case leverages program analysis and constraint solving to explore the target. The use case of these three can depend on the available resource at hand or the type of application. It is possible to achieve better fuzzing results by combining approaches.

There are two widely used methods for fuzzers to generate test cases: mutation-based and generation-based (also known as generator-based or grammar-based) [33, p. 1202].

A mutation-based fuzzers are typically general purpose fuzzers [34, p. 137]. A mutation-based method requires a well-formed baseline (e.g. a single or corpus of seeds) as base to mutate and produce inputs [33, p. 1202]. The mutation can be performed using a variety of methods (e.g. randomly or predefined mutation strategies which can be adjusted based on gathered information related to the target during run-time). The efficiency of the fuzzing is reliant on the quality of the baseline. A good corpus must be generated in some way, especially for more complex targets. Since a mutation-based fuzzer does not understand the underlying format it is working with, another challenge arise when, for example, dealing with blocking components such as checksum checks, encryption/decryption, authentication, etc. If not handled correctly, they often result in an exception, and the fuzzer is unable to exercise the code after it. Often these can be solved by manually stubbing these components.

On the contrary, generation-based fuzzers are often specialized fuzzers for a specific target (e.g. application, protocol, etc.) [34, p. 137]. The generation-based method does not require a seed or corpus [33, p. 1202]. Instead, it generates inputs from scratch by modeling the target protocol or file format when provided a specification (e.g. formal grammars, file formats, network protocols, etc.) of the target. This approach requires much manual work on defining the specification of the target for the fuzzer.

There is not clearly superior method to fuzz. It depends greatly on the resources at hand and the nature of the application. For example, when using a mutation-based method, it is possible to fuzz without having a specification of how a protocol works, or an API description of a certain function. It is able to fuzz many different type of applications, with little to no knowledge about it. Sometimes, seed modifications at a lower level can lead to the finding of different bugs, which testing with a well-defined input model cannot find. In contrast, generation-based fuzzing can often provide fully valid test cases to blocking components in the target. The fuzzing can more easily exercise all parts of the target.

The fuzzing can sometimes be considered as "smart" and "dumb" [34, p. 144]. A smart fuzzer has information about the target and utilizes this in a way that improves the fuzzing efficiency, unlike a dumb fuzzer. For example, a mutation-based fuzzer might modify seeds randomly. However, it might receive information regarding code or path coverage and utilizes this to keep modified test cases that exercise new uncovered areas of the target, keeping them as references for further modification. This mutation-based fuzzer can be considered smart. While, in the case of a generation-based fuzzer, it might understand the input model. However, it might just blindly send test cases without considering any feedback information to improve the fuzzing process. This can be regarded as dumb fuzzing.

3.6.3 Fuzzing with AFL and Sanitizers

American fuzzy lop (AFL) is a well known open source fuzzer which will be used in this project to perform security testing of TAs [35]. The AFL project is not a proof of concept, but rather a bunch of hacks. AFL is a coverage guided evolutionary fuzzer. It supports both grey-box and black-box fuzzing.

The initial AFL project in Google's repository has been relatively inactive, leading to a fork of the initial project named AFLplusplus. This is project is community-maintained and includes new features and speedups [36, 37].

3.6.3.1 Architecture

The architecture of AFL is illustrated in figure 3.7 [38, p. 242]. The main component of AFL is fuzzer, which performs the actual fuzzing. The fuzzer will start by loading initial inputs provided by the user into the seed queue [39, p. 2]. If the forkserver mode is enabled, a forkserver will be used to spawn new instances of the fuzzing target [36]. Input from the queue will be used at each run of the target. During a run, feedback information will be sent back to the fuzzer. This information will provide the fuzzer with coverage and crash information allowing the fuzzer to generate new input through mutation for the adjacent runs and allowing it to store useful inputs (e.g. covering new execution paths) back into the queue [39, p. 2].

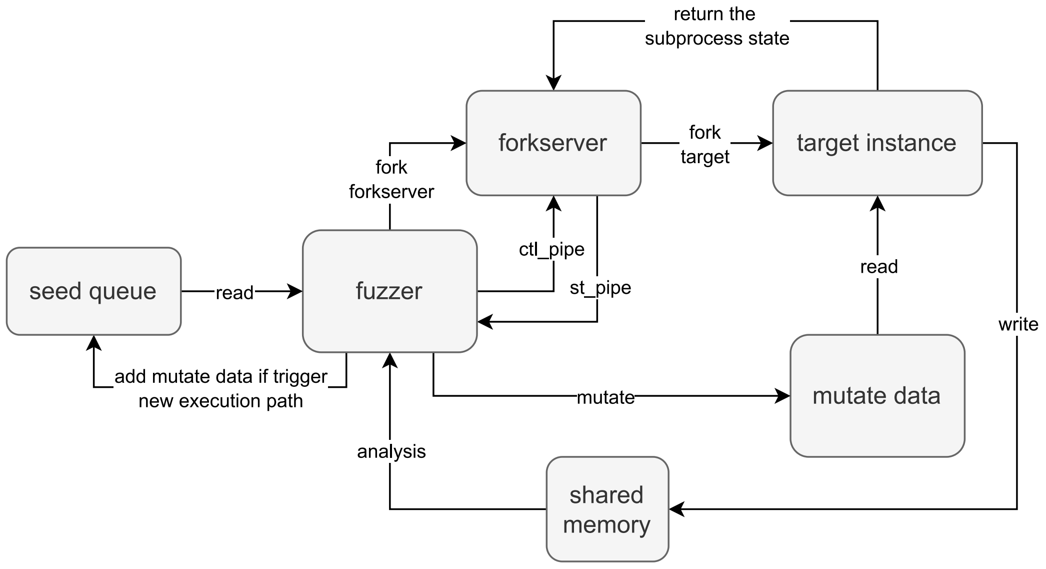

Figure 3.7: Overview of the AFL architecture.

3.6.3.2 Mutation

Different fuzzing strategies are used by AFL to generate new inputs to the target. Some of these strategies are more expensive than other[40]. The strategies are divided into deterministic and non-deterministic [39, p. 37]. The fuzzer mutates the inputs using the different strategies in a specific order, starting from the deterministic strategies as they tend to produce simple and compact test cases and smaller differences between a crash and non-crash yielding input. An exception to this is if the fuzzer notices that the initial deterministic strategies on a region of the input do not have an effect on the execution path checksum it will skip the remaining deterministic strategies and start mutating inputs using non-deterministic strategies instead. The deterministic strategies include *walking bit flips*, *walking byte flips*, *simple arithmetics* and *known integers* [40]. The non-deterministic strategies include *stacked tweaks* and *test case splicing.*

3.6.3.3 Instrumentation

Finding test cases that can attain a full coverage on a relative large and complex program is usually unfeasible. Instead, tracking the progression of fuzzing and make use of it in the fuzzing process can be a substantially important factor in efficient fuzzing. AFL do this by measuring code coverage. Specifically, AFL captures branches (edges) and counts coarsely the number of times they have been exercised by the target application during execution [39, p. 33]. This is done by performing instrumentation on a low level (binary). This can be performed, assuming the source code is available, by inserting the instrumentation (additional assembly instructions) during compilation of the target application at smart locations such as before if cases and function calls. When these instruction are executed, routines are called to collect process information for the fuzzer. AFL utilizes tools such as *clang* or *GNU compiler collection (GCC)* for injecting instrumentation into the target [39, p. 3]. This instrumentation is light weighted and has a moderate performance impact.

Another possible approach, assuming the binaries are available, is to perform the instrumentation during emulation of the target application. In the AFL case, there is a *user emulation mode* that uses a customized QEMU, an open source full-system emulator, to emulate the target and to receive instrumentation information from the emulation. *Basic blocks* are continuous instructions without branches except for its entry and exit point. QEMU uses basic blocks of the target as translation units

3.6.3.4 Increase performance

The fuzzing efficiency is increased in AFL by injecting a shim, a small piece of code, into the target during the instrumentation [41]. The shim acts as forkserver and will be used to bypass initialization of the target such as the linker. The forkserver will be signaled by the fuzzer to fork the target process, creating a copy of the already initialized process. Hence, saving time. This will allow faster fuzzing throughput, since a new process is created for every new input file. At minimum, an increase in performance is made already by starting the shim at the main method. This can give a performance gain between 1.5-2 times faster [39, p. 38]. The injection point can also be manually controlled to bypass target specific initialization. This is referred to as *deferred mode*. However, it should be inserted before the input vector to be tested, and before pipes, character devices, sockets. etc. since they are I/O that cannot be reset. In some target, performance can increase over 10 times.

In AFL there is *persistent mode* specifically designed for stateless API targets [42]. The requirement is that the target must be able to reset its state completely between function calls to avoid resource leaks, as any prior runs will not impact the latter. This mode allows a single process of the target to be reused, instead of having OS overhead from forking it. It is also possible to utilize shared memory for further increase in performance. Instead of receiving input test cases via stdin or files it can be delivered to the target via shared memory.

3.6.3.5 Utility

Utility tools are included in AFL. For example, AFL takes an input file (seed) or a corpus before the fuzzing begins. These are examples of actual input data to the target. In a corpus, there can be inputs that exercise same functionality. A tool *afl-cmin* can be used to generate a subset of the corpus with input files that exercise different code paths.

Another useful AFL utility tool is *afl-cov* [43]. It will display coverage information in a human-readable format e.g. which code lines have been reached and the number of executions performed on a certain line, allowing the user to evaluate the fuzzing harness and input seeds. afl-cov uses *gcov* instrumentation to generate the profiling information of the target [44]. It then uses *lcov* to generate a front-end presentation of the gathered profiling information in form of a navigatable website [45].

3.6.4 Sanitizers for Improved Bug Detection

Sanitizers are used for run-time detection of behaviour in an application. There are different sanitizers are available in the *LLVM* and GCC toolchain. When used in a fuzzing context, sanitizers are able to improve detection capabilities of bugs, but will have an impact on the performance of the fuzzing. They are often used in fuzzing contexts. Despite the compromise in speed, enabling sanitizers often results in better fuzzing results. A variety of sanitizers can be used with fuzzers.

The *AddressSanitizer (ASAN)* allocates big regions of virtual memory for bookkeeping [39, p. 23]. Unless the virtual memory is accessed, the OS will not allocate physical memory. This can allow fuzzers e.g. AFL to detect if the target runs off rail. ASAN is a powerful tool, and is able to detect a variety of problems [46]:

- Use after free (dangling pointer dereference)
- Heap buffer overflow.
- Stack buffer overflow.
- Global buffer overflow.
- Use after return.
- Use after scope.
- Initialization order bugs.
- Memory leaks.

Other sanitizers can also be useful to find faults in a target. For example, *LeakSanitizer (LSAN)* can be used to detect memory leaks [47]. This is used by default when ASAN is enabled. However, it does not require ASAN instrumentation and can be used in stand-alone mode.

MemorySanitizer (MSAN), which detects uninitialized memory reads performed by the target [48]. This happens when a memory in the stack or heap has been read before written. MSAN is used to detect such behaviour and generates a crash if it affects the program execution.

Sanitizers (e.g. ASAN, MSAN, etc.) causes incompatibility issues with the QEMU user emulation. Consequently, AFL in user emulation mode is unable to run with sanitizers that utilizes such technique, meaning that sanitizers work only if compile-time instrumentation is an available option.

Another sanitizer, *UndefinedBehaviorSanitizer (UBSAN)*, detects undefined behaviours of an application [49]. It is able to detect:

- Array subscript out of bounds, where the bounds can be statically determined.
- Bitwise shifts that are out of bounds for their data type
- Dereferencing misaligned or null pointers.
- Signed integer overflow.
- Conversion to, from, or between floating-point types which would overflow the destination.

There are many more sanitizers available. Factors such as target architecture, the fuzzer and available resource will determine if they can be used.

3.7 PKCS#11

The PKCS#11 TA running in OP-TEE is a relatively complex application and implements a widely used standard. It can therefore potentially be used with many devices. Hence, it makes a good target to test for potential vulnerabilities. An implementation of it can be enabled and used in OP-TEE.

3.7.1 The PKCS#11 Model

PKCS#11 is an API (sometimes referred to as Cryptoki) that handles the communication to cryptographic devices known as *security tokens* or just tokens [50]. Figure 3.8 illustrates how an application uses the Cryptoki interface to send a request to a token [50]. These tokens can represent a logical view of devices that perform cryptographic operations and store objects.

Examples of what a token could represent is a smart card, USB key, HSM, etc. Tokens are contained in a slot. A slot represents a physical device interface or a logical reader. This could be, for example, a smart card reader.

There can be multiple slots and each slot can contain multiple tokens, performing various cryptographic operations with those. It is even possible to share common tokens between slots. Objects stored in a token are generally divided in four classes:

- *Data objects.* These are objects defined by an application.
- *Certificate objects.* These are digital certificates such as X.509[2].
- *Key objects.* These can be private, public or secret cryptographic keys.
- *Vendor-defined objects.* These are general objects, whose representation are decided by the vendor.

An object can be a *token object* or a *session object.* A token object can be accessed by any application with the required access permission. These tokens are persistent and destroyed only when a token has been specified to do so. In contrast, a session object is specific to a session. They are created when a connection is made from an application (e.g. a CA) to the token. The connection is referred to as a session. Session objects are created upon the opening of a session and destroyed when the session is closed. The created session objects can only be viewed by the application that created it.

The access to objects are controlled by their visibility level. *Public object* are visible to all applications, meanwhile *private objects* are only visible to the application, if the user is logged into the token. There are two types of users: a *security officer (SO)* and *normal users.* The SO has an administrator role. A SO performs only two things: initialization of tokens and configuration of normal user PINs.

3.7.2 PKCS#11 in OP-TEE

As tokens contain security sensitive information and operations the interface to them must be proper and protected adversaries. Hence, the PKCS#11 is implemented in the form of a TA protected in the secure world.

[2]A standard defining the format of public key certificates `https://datatracker.ietf.org/doc/html/rfc5280`.

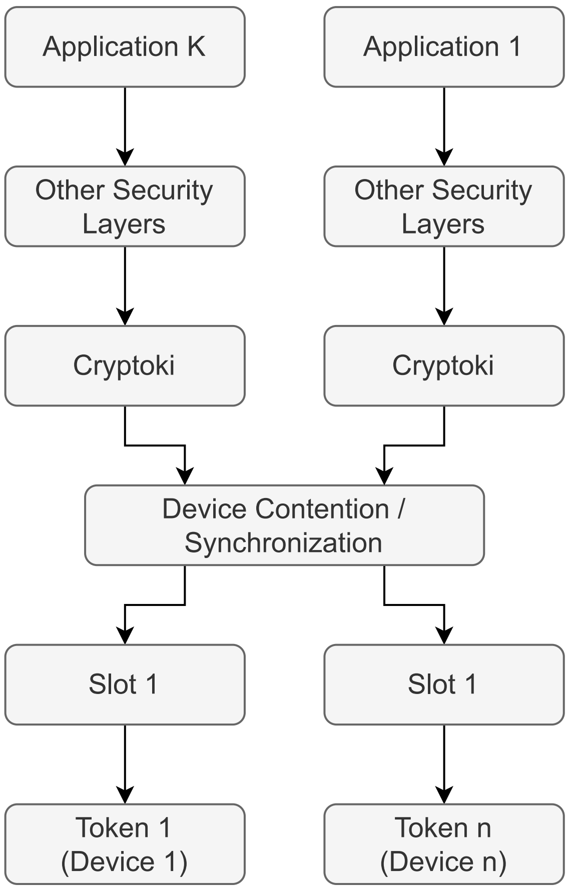

Figure 3.8: The figure illustrates the general PKCS#11 model

In OP-TEE, the PKCS#11 API is implemented with the *libckteec* client library and the PKCS#11 TA as shown in figure 3.9 [23]. The libckteec reside in the normal world and implements the PKCS#11 API on top of the TEE client API implemented in form of the client library *libteec*. The libteec library is used by a CA to communicate to the secure world TA. Every PKCS#11 operation is mapped to a command ID and implemented as a single command invocation using the four parameter provided in the TEE client API. Additional parameters or data can be provided to the PKCS#11 TA using shared memory and serialization. On the secure side an in-tree TA is implemented to act instead of a HSM to perform secure operations (e.g. cryptographic operations, key management, secure storage, etc.). It resides in the optee_os source tree. PKCS#11 can be included in OP-TEE by setting the *CFG_PKCS11_TA=y* before build.

3.7.3 Passing Data to the PKCS#11 TA

Serialization are performed on Cryptoki objects before passing them to the TA using shared memory in order to circumvent the limited number of memory references allowed to be passed by the GP API. The *serialize_ck_attributes* function is used by the CA to serialize the data, storing the data in a shared memory buffer which will be passed to the TA. And if the TA returns updated serialized data, the CA calls the *deserialize_ck_attributes* function to deserialize the returned data via shared memory allocated by the CA. The deserialization uses the CA provided input arguments to determine the serialized buffer structure, and in combination with the actual returned buffer the function is able to decode the value, size and type of the returned data. Such objects can be CK keys, certificates, mechanism parameters, etc.

3.8 Related Work

3.8.1 Android TA Fuzzing

There is paper that introduces a new fuzzing framework, TEEzz, which enables fuzzing on TAs on commercial Android devices [51]. The target TA is essentially a black-box with exposed API, whose input format and states are unknown. TEEzz performs on-device fuzzing based on AFL++ by injecting inputs via the driver interface. As black-box model targets only

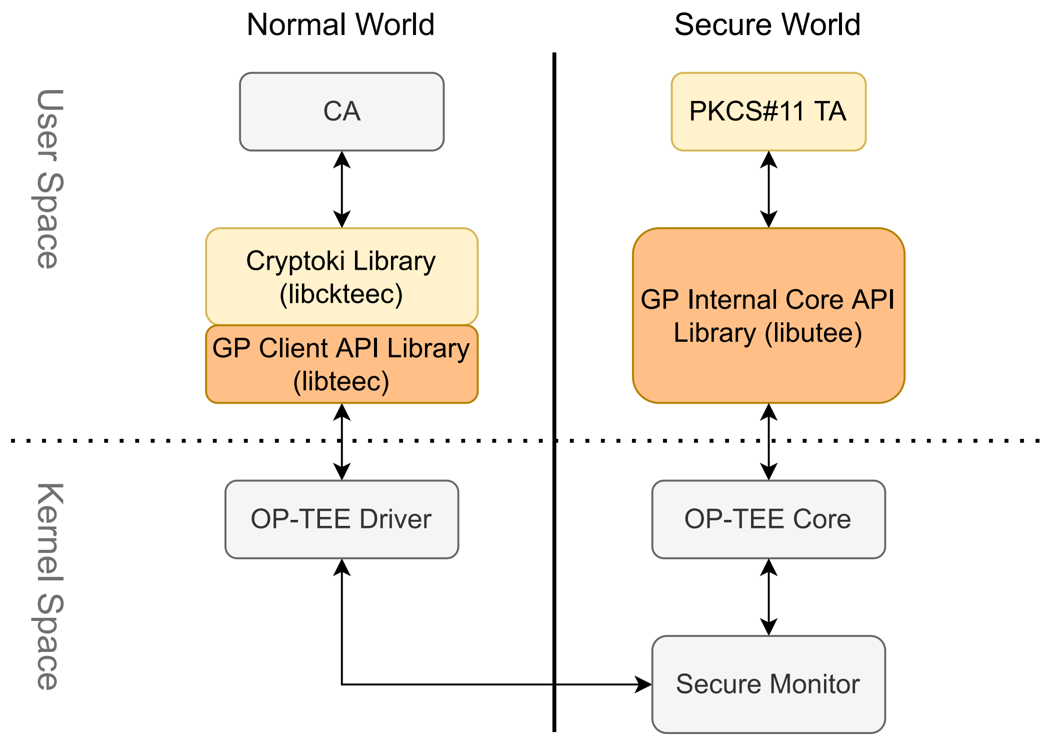
Normal World
Secure World
User Space
Kernel Space
CA
Cryptoki Library (libckteec)
GP Client API Library (libteec)
OP-TEE Driver
PKCS#11 TA
GP Internal Core API Library (libutee)
OP-TEE Core
Secure Monitor

Figure 3.9: The figure shows an overview of how PKCS#11 is implemented in OP-TEE. The yellow components are the implementation of PKCS#11 in OP-TEE while the orange components are the implementation of the GP API.

reveal the binary, type- and state-aware fuzzers cannot be leveraged without access to the source code or some form of logging. Instead, to improve the number of valid fuzzing data that passes the SMC interface and reaches the TA, TEEzz uses automatic data type deduction by combining dynamic binary instrumentation, multi-interface interaction recording, and semantic deduction. To have a stateful fuzzing, dynamic binary instrumentation is used to record the interaction between the CA and TA to attain data dependencies which can be used to build a model of the different call sequences for the TA and the corresponding input data. This can then be used to infer the API model of the TA to efficiently fuzz it in different states. TEEzz then tries to combine the information to find value dependencies between different interactions in order to produce useful seeds for the fuzzer. Finally, it passes the seeds to a CA specific mutator to fuzz the target.

The fuzzing framework focuses on on-device fuzzing of TAs in commercial-off-the-shelf Android devices in a black-box fashion by gathering information by recording the interactions of binaries and then tries to deduce useful information for fuzzing. In contrast to the TEEzz project, this thesis will rehost the TA through emulation and not be dependant on specific hardware. All the source code is also assumed to be available which allows more sophisticated fuzzing techniques and assisting tools. An advantage of grey-box/white-box fuzzing over black-fuzzing is the increase code coverage and the possibility to fuzz any particular parts of the code (e.g. critical parts) without resolving to any reverse engineering or semantic deduction. A difference to note, is that grey-box/white-box fuzzing might be disadvantageous in a certain aspect in comparison to black-box fuzzers, even though the source code of the application is available, the grey-box/white-box fuzzers require lots of extensive research into the application's source code, which is written in a specific programming languages, which makes it harder to adapt to other applications. In comparison, black-box fuzzers are more adaptable across applications as they are input/output problems usually without dependency on the application's specific programming language [52].

3.8.2 Qualcomm TA Fuzzing

Another work focuses on fuzzing TAs running in Qualcomm's Secure Execution Environment by building a feedback-based fuzzing platform [53]. The target is considered a black-box, similarly to the case of the Android TA fuzzing case. The idea here is to use a loader to receive data- and code segmentation dumps and virtual addresses of the target TA in the secure world to execute the TA in the normal world. Syscalls in the normal world TA is handled by redirection, using QEMU syscall handler, to the TA in secure world. In the secure world, the signed target TA needs to be patched. As the TEE used comes with a secure boot that cannot be disabled in a legitimate way, two known exploits is chained to break the verification mechanism in order to load a patched TA into the TEE. The actual fuzzing of the emulated TA command handler is performed by using AFL++ in user emulation mode that is able to produce coverage metrics using the QEMU emulator.

The solution focuses solely on the Qualcomm environment and relies heavily on existing vulnerabilities of the target system. Therefore, similar solution cannot easily be produced for other TEEs and TAs. The solution also has a big performance overhead due to the QEMU emulation, context switching between the two worlds, additional operations performed by the loader and handling of syscalls. Using the AFL++ in user emulation mode also has other limitations such as the usage of sanitizers for enhanced bug detection.

System Design

In order to implement a proof of concept of TA fuzzing with source code, different design decisions had to be made. This includes what environment to run the TA in during the fuzzing; how the migration of the TA to the new environment will be done; deciding on tools to use for the fuzzing; how to design the harness for the target. These will be motivated in this section.

4.1 The Fuzzing Environment

There are requirements for running an effective fuzzing session. The solution for the project will include designing an environment able to fulfill these conditions. The core requirements identified will be presented and some ideas and the final solution will be discussed below.

4.1.1 Requirements

A TA is normally ran inside a TEE on a hardware e.g. an embedded system. However, it is possible to emulate a native TEE and hardware in form of software. This allows for a flexible solution independent of specific hardware and can be used for development purposes. In order to fuzz a TA running inside a TEE in such an environment, certain criteria imposed by effective source-code-utilizing fuzzing techniques should be met:

- The fuzzer is able to run alongside the TA.
- The fuzzer is able to invoke and relay input data (e.g. generated or mutated from the fuzzer) to the TA process.
- Some TA process related information (e.g. branch/edge information, code coverage information, process information, etc.) has to be conveyed to the fuzzer.
- The TA source code is available and the TA binary can be built.

4.1.2 Ideas and the Solution

Four ideas of possible solutions that came up in the brainstorming session are presented here. The main problem of each solution will be shortly discussed.

4.1.2.1 Fuzzer as a TA

A TEE is designed for certain architectures. In the case of OP-TEE it is designed for ARM architecture and would normally work with an ARM processors with the TrustZone security extension. However, QEMU can emulate the native environment, creating a normal and secure world separation allowing OP-TEE to function properly on other architecture than ARM (e.g. x86). In this approach the fuzzer will be built as a TA running in OP-TEE in the secure user space (S-EL0) as depicted in figure 4.1.

The idea is that the fuzzer relay inputs to the target TA via the GP TEE internal API using the TA2TA interface. This interface allows for inter-TA communication directly in the secure world. The main problem of this solution is that the fuzzer would have to rely on functionality offered by the OP-TEE core. Common fuzzer solutions are often dependent on a variety of common REE syscalls such as fork, execve and open that are not implemented in OP-TEE. Any calls to those will result in a *TEE_ERROR_NOT_IMPLEMENTED* error being emitted. To use this approach, these missing syscalls would have to be implemented rendering this approach ineffective.

4.1.2.2 Fuzzer as a normal world application

To avoid the problem of missing syscalls in the previous solution, the fuzzer will instead be built as a normal user space application in the normal world (EL-0) in a QEMU emulation as depicted in figure 4.2. This will allow the fuzzer to perform common syscalls. As OP-TEE is designed for Arm architectures, the fuzzer would also have to be built for Arm. The idea is that the fuzzer relay will use a harness (CA) which will invoke the TA with the test cases. The fuzzer will then need coverage and process information of the target TA for efficient fuzzing. In OP-TEE, TAs can be instrumented with gprof instrumentation to generate profiling information. Gprof is used for performance analysis and can collect coverage information to generate call graphs and display the number of executions that has been performed in

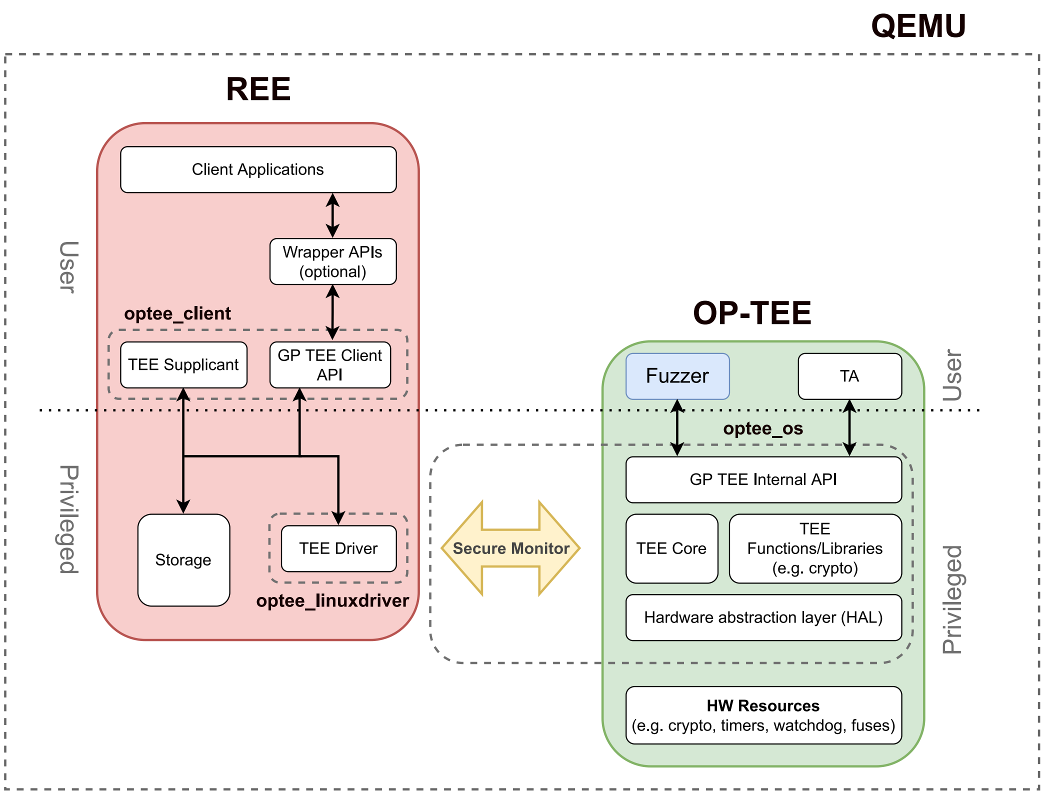

Figure 4.1: Running fuzzer as a TA in secure world.

specific areas of a binary [54]. This information could perhaps be transformed into a format used by the fuzzer. To provide the fuzzer with additional information of application misbehaviour, sanitizers will be used. This is often achieved with sanitizers i.e. ASAN which can detect memory corruption, dangling pointer, etc. The main problem of this solution is that in OP-TEE only the core can be built with ASAN, however this feature is absent for TAs. Without the usage of ASAN, the fuzzer's ability to detect misbehavior of the target would be limited. This information is essential for fuzzers to detect many different types of bugs in the target

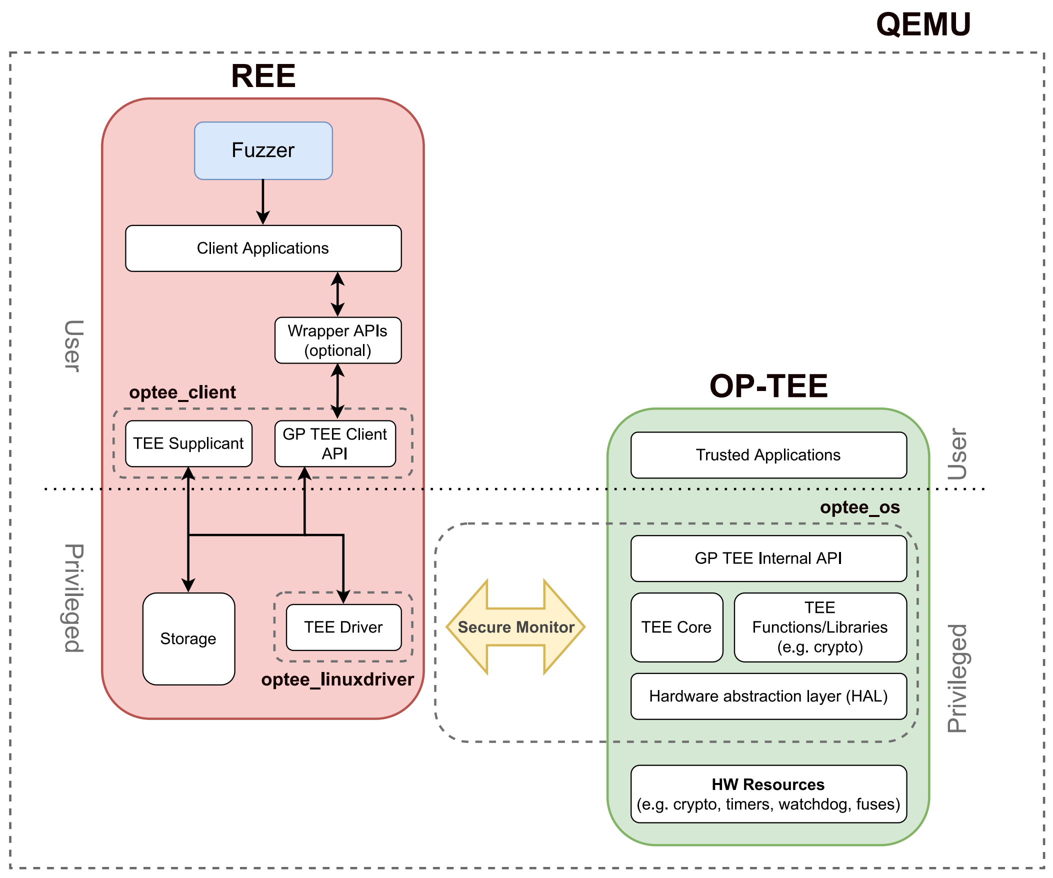

Figure 4.2: Running fuzzer as a normal world application.

4.1.2.3 Proxy TA in secure world

This hybrid solution combines the idea of the previous two solutions. A QEMU emulation will be assumed. The fuzzer and target TA will be running as user space applications in the normal world (EL-0). Any OP-TEE specific syscalls will be handled by a TA running in the secure world user space (S-EL0) as depicted in figure 4.3. This will allow the target TA in the normal world to be built with coverage and sanitizer instrumentation providing the fuzzer with the necessary information for an effective fuzzing session. The pitfall of this solution is that any OP-TEE syscall turn into a library call to the GP API that will have to go through the TEE driver, secure monitor and OP-TEE core before it can invoke the TA to perform the syscall. The result of the syscall will have to be returned back to the TA running in normal world. This is a major performance overhead and will affect the fuzzing performance, which will in turn lower the efficiency of the fuzzing.

4.1.2.4 Emulate the GP TEE Interface

This idea will not be based on any QEMU emulation and there will not be any world separation (REE/TEE) as in the previous solutions. Instead, the target TA will be running directly on the host OS using a virtual TEE emulator that is implemented using conventional OS features on common architectures. The virtual TEE needs to be able to handle all the TEE specific syscalls, providing the necessary GP APIs and other functionalities the TA is dependent on. The context switching between the normal and secure world will be replaced by IPC via the TEE emulator as depicted in figure 4.4.

To realize this, a project Open-TEE will be used. Open-TEE is an open source hardware-independent TEE implemented in software [55]. It is designed to run in the GNU/Linux environment and relies on technologies and services offered by these mainstream OSes [25, 24, p. 6]. The Open-TEE framework also uses open-source software e.g. the *Mbed TLS (Mbed Transport Layer Security)* crypto library for crypto operations and the GNU tool suite. The goal of the project is to simplify the development process of TAs for more developers, allowing them to develop and test TAs using familiar tools and environment [24, p. 1]. Open-TEE allows a TA to be ran on a x86 architecture and can benefit from higher performance with more powerful hardware, less overhead and higher CPU utilization. Many fuzzer are designed for mainstream OS such as GNU/Linux and is therefore able to operate alongside the TA. Instead, the

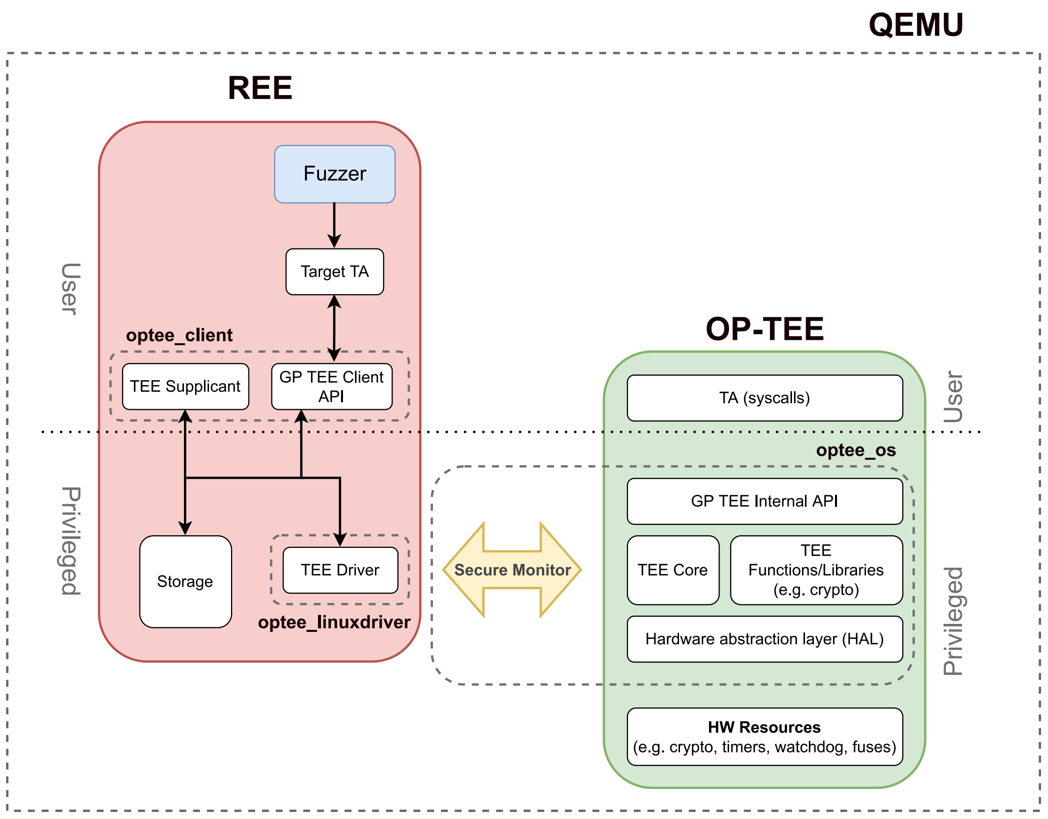

Figure 4.3: Running the target TA in normal and secure world to handle syscalls.

challenge has shifted to porting the PKCS#11 TA to this new environment. This feasible approach will provide a more generic solution than the previous ideas as the engine is designed to emulate a generic GP conforming TEE. It also allows the usage of common tools of a mainstream OS and can be ran on architectures such as Arm and x86, allowing the usage of more powerful hardware to accelerate the fuzzing process. As this solution is not based on a TA native or near native TEE, the target TA behaviour might deviate and any TEE specific feature dependencies will also have to be resolved. The Open-TEE project claims to have fully implemented the GP internal core, but having a 80% cryptographic algorithm coverage as the libraries in use lack support for the remaining algorithms [24, p. 6].

This approach trades accuracy for genericness (not TEE specific and compatible with more fuzzers), with additional benefits of performance and flexibility (more tools available). For the many benefits of the approach, this will be used in the final solution.

4.1.3 Porting of PKCS#11

Now that it is decided on how the target TA will be hosted, the PKCS#11 TA have to be ported from OP-TEE to this new environment. TAs are designed to conform with the GP internal core specification, meaning they should not be bound to any specific TEE implementation that is GP compliant. The PKCS#11 TA is designed to communicate with a CA using Cryptoki, the PKCS#11 API library, in OP-TEE as shown in figure 3.9. A similar implementation of PKCS#11 is provided in the Open-TEE project and could possibly be used to simplify the the porting of the target. Upon reviewing the TA and library provided by Open-TEE, some major differences are identified. The TA was found to be lacking in functionality compared to OP-TEEs implementation, and the PKCS#11 internal serialization logic implementation differed in how the data in the allocated shared memory buffer are prepared before passing the GP API. Serialization and deserialization is performed in both the Cryptoki library and the PKCS#11 TA for many of the available TA invocation. As the two implementation of PKCS#11 are incompatible, the Open-TEE implementation is not used. Instead, both the PKCS#11 TA and Cryptoki library found in OP-TEE has to be ported to function as it was intended. It was also found that OP-TEE provide extensions to the GP internal core API which the PKCS#11 TA is reliant on [6, p. 51]. Therefore,

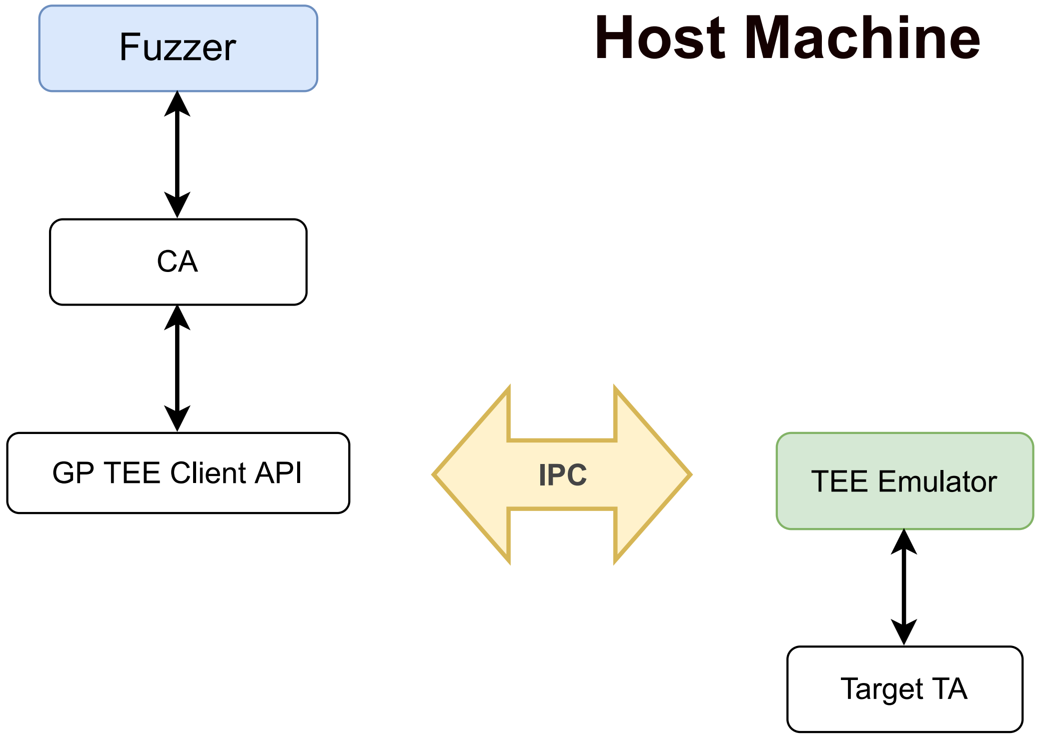

Figure 4.4: Running the target TA directly on host OS.

additional functionality needs to be ported while other less important feature can be stubbed out in order for the TA to work with the Open-TEE engine. The Cryptoki library (*libckteec* in OP-TEE) will be built on top of *libtee*, Open-TEE's implementation of the GP client API and the PKCS#11 TA will have to use functionalities provided in *libInternalApi*, the GP internal core API provided by Open-TEE. Finally, the GP version that is implemented in OP-TEE and Open-TEE differs and not fully compatible. The mismatch in the API assumed by the OP-TEE ported code and the provided API in Open-TEE resulted in some bugs that is solved by adjustments to the API for consistency.

4.1.4 Binary Instrumentation

Different instrumentation will be performed on binaries during the compilation phase. The CA and TA binary will be instrumented with AFL instrumentation to spawn the forkserver during a fuzzing session and to enable a feedback based fuzzing process. Additional gcov instrumentation for coverage information and ASAN instrumentation for enhanced bug detection will be performed on the TA binary. The ASAN is configure in the TA to dump any crash report to a file on the file system for later analysis. The gcov instrumentation will provide the user with additional coverage information to the user.

4.2 Modern Fuzzing Tool Alternatives

Next, a fuzzing tool must be chosen to perform the actual fuzzing. There is a plethora of fuzzing tools, each designed for different purpose in mind. Some may be very specialized tools for a certain environment e.g. kernel, web etc. while other are designed for general purpose fuzzing. There are even fuzzing frameworks for designing a tailored fuzzer for a specific task. An extract of well-known fuzzers are presented in table 4.1 [8, 56, 57, 35, 58, 58, 59, 60, 61].

Table 4.1: Well-Known Fuzzers

Name	**Targets**	**Key Technique**	**Platforms**	**Availability**
SAGE	Large Windows applica-tions	White-box generation fuzzing	Windows	Microsoft internal
Syzkaller	OS kernels	Coverage-guided grey-box mutation and generation fuzzing	Linux	Open source
vUSB	USB drivers	Input knowledge based black-box generation fuzzing	Linux	Open source
AFL	General Purpose	Coverage-guided grey-box fuzzing, genetic algorithms	Linux, OpenBSD, FreeBSD, NetBSD, MacOS, Solaris	Open source
FairFuzz	General purpose	Coverage-guided grey-box mutational fuzzing	Linux, OpenBSD, FreeBSD, NetBSD, MacOS, Solaris	Open source
honggfuzz	General purpose	Evolutionary, feedback-driven fuzzing based on code coverage	GNU/Linux, FreeBSD, MacOS, Android	Open source
libFuzzer	General purpose	In-process, coverage-guided, evolutionary fuzzing	Linux, MacOS, Windows	Open source
Peach	General purpose	Black-box mutation and generation fuzzing	Linux, Windows, MacOS	Open source

4.2.1 Evaluation of Fuzzing Tools

Each fuzzer is designed with different use case and has both pros and cons in different fuzzing scenarios, making it difficult to grade a fuzzing tool. The result could greatly vary depending on the application domain, input structure, the actual operation performed etc. In most cases they are exercised in different benchmark tests with known or unknown bugs. Common evaluation metrics used in these benchmarks are the percentage of code/branch coverage, number of crash triggered and the number of bugs found [62, p. 1].

Externally performed fuzzer benchmarks will be used to decide on a fuzzing tool to perform the fuzzing on the PKCS#11 TA for the aforementioned reasons. Bug detection capability and code coverage based tests will be prioritized to be used in our evaluation. Only fuzzers that uses source-code-utilizing fuzzing technique from the benchmark results will be considered a candidate. Below two external results from testing different fuzzers using two different benchmark tools for fuzzers. The first benchmark tool will focus on the most important metric of a fuzzer: the number of bugs it finds. However, the project also focuses on utilizing the source code of the target to achieve better fuzzing. The additional information provided to the fuzzer can allow the fuzzer to increase the fuzzing coverage of the target, allowing more bugs to be found. In the second benchmark tool, the coverage exercised in the fuzzing targets will be used as a metric in the results, providing a better evaluation of the fuzzers.

4.2.1.1 Magma

Magma, a fuzzer benchmark focused on using ground truth bugs, was used in [62] on the following set of widely used mutation based fuzzing tools for evaluation: AFL, AFLFast, AFL++, FairFuzz, MOpt-AFL, honggfuzz, and SymCC-AFL. The different fuzzing tools were exercised over 200,000 CPU-hours using the same set of seeds on applications with bugs and were evaluated based on the number of bugs reached, triggered and detected. The test targets used in Magma are seven applications with widespread use in real-world environment containing diverse and varied security bugs that have been reported throughout their lifetimes. A total of 118 bugs are reintroduced in the test targets for benchmark purpose and are accompanied by an oracle that is able to report if the bug was reached or triggered by the fuzzer. In the benchmark tests AFL++ was found to score the highest mean bug count in comparison to the other fuzzers and that this difference is statistically significant [62, p. 16]. However, other fuzzer were shown to perform better on certain test targets.

4.2.1.2 FuzzBench

FuzzBench, is a benchmarking service for fuzzers, an alternative to Magma [63]. The projects focuses on simple and fast test integration for fuzzers, and claims to be integrable with around 100 lines of code. It utilizes Google's computer resources for benchmark runs [63, p. 1394]. Unlike Magma, which focuses on measuring the number of bugs found in its evaluation, FuzzBench can use code coverage and bug discovery. If focusing solely on bug discovery without code coverage in the evaluation, the result can be misleading and biased as real-world bugs are often many and sparse in a big code base [63, p. 1395].

In order to use the service, the user will have to integrate a fuzzer, which they want to benchmark, with the FuzzBench API to allow the fuzzer to build and fuzz FuzzBench benchmarks e.g. any OSS-Fuzz target. After deploying the tests, FuzzBench will produce reports comparing the performance of the fuzzer in test to other popular fuzzers. The reports will also include statistical confidence intervals which can be used for further analysis. This service is popular among fuzzer developers and is frequently used to improve fuzzers e.g. honggfuzz, libFuzzer and AFL++ [63, p. 1397]. An evaluation of 11 fuzzers were conducted in [63] using FuzzBench. This benchmark included the following fuzzers: AFL, AFLFast AFL++, AFLSmart, Eclipser, Entropic, FairFuzz, honggfuzz, libFuzzer, MOpt-AFL and lafinte. They were benchmarked against 22 different open source project from the OSS-Fuzz default benchmark set. The targets represented a wide range of user space program, taking a variety of input formats e.g. XML, JPEG and ELF. Some tests are provided seed and dictionaries, while other are not, to better reflect real world scenarios as these might be absent. The total run-time of approximately 111 320 CPU hours was performed. The benchmark resulted in AFL++ having the highest average rank, followed by Honggfuzz, Entropic, and Eclipser [63, p. 1397]. The report also showed that there were no statistically significant difference between the seven highest ranking fuzzers.

4.2.1.3 Choosing a Fuzzer

As previously mentioned, it is a difficult task to evaluate and find the optimal fuzzing tool for a specific task at hand. Due to time limitations of the project, it will not be possible modify existing fuzzers for optimization or to craft a specialized custom fuzzer using a fuzzer framework. Specialized fuzzers for other specific tasks are excluded in our evaluation since they are designed for other use-cases in mind. The PKCS#11 TA, is a fairly big and complex application in itself, which will require heavy manual specification of the input model for a generation-based fuzzer. For this reason smart mutation-based fuzzers are preferred over generation-based. Another preference is for the fuzzing tool to be open source as modifications to the tool could be required to enable fuzzing with the environment setup.

After reviewing the result of the two different fuzzer benchmark tools tested on multiple popular general purpose fuzzers, AFL++ was found being the strongest competitor among the evaluated fuzzers when prioritizing the average bugs discovered and coverage covered in a target. In the Magma benchmark AFL++ was able to discover the most ground truth bugs in average among the mutation based fuzzers, and in the Fuzzbench benchmark it scored the highest considering both the number of bugs found and coverage covered. While many fuzzers can perform better on certain targets, AFL++ is performing well in a variety of different targets in both bug discovery and coverage making it a safe choice. In addition, AFL++ is mutation based, ready to use fuzzing tool, allowing for easy deployment. AFL++ is often mentioned in many papers and conferences related to fuzzing and has an active community driving its development. It also uses grey-box fuzzing technique allowing the utilization of TA source code and documentation available. For the aforementioned reasons, AFL++ has been chosen to perform the fuzzing on the PKCS#11 TA. Any mentions of AFL will be a reference to AFL++ in the remaining parts of this thesis.

4.2.2 AFL Integration

After porting the PKCS#11 TA to Open-TEE, and AFL has been selected as the fuzzing tool to use, the fuzzer has to be integrated to work with the setup. The AFL model, as shown in figure 3.7, expects to instrument and spawn the target binary. In the case of GP compliant TEE, the CA always initiates the TA making it the target to be forked by the instrumented forkserver. However, the fuzzing target is instead the TA that it invokes and communicates with. Due to the deviations in the AFL model and the CA-TA relation, modification has to be performed in AFL. In order to make AFL use edge information generated from the instrumented TA, it has to receive the process identifier (PID) and process status information of the TA instance. The PID is used by afl_fuzz to monitor the target process and detect when a crash has occurred. Hence, the PID and status of the TA has to be relayed to afl_fuzz. This can be done using inter-process communication (IPC), to reflect how the process information is normally relayed from a target to AFL. In order for AFL to receive TA process related information, modifications to Open-TEE and AFL is made. The communication between them uses a named pipe. This is a design choice made to be similar to anonymous pipe, what AFL natively uses to handle communication. AFL receives the TA PID from the launcher process in Open-TEE during TA process creation, and status update from the manager process when either the TA raises a SIGCHLD signal or exits successfully. An overview of the relationship between AFL, Open-TEE and the target TA is shown in table 4.5.

The CA will be used to create the harness to the target. AFL persistent mode is used on the CA for performance benefits, allowing the CA process to be reused to avoid reinitializing the process and reloading libraries again. A new TA will however be created after a command invocation followed by closing session and finalizing context. The Open-TEE engine is ran as a daemon process in the user space and must be ready to accept TA invocations from CAs. However, the implementation of the Open-TEE is rather unstable. It is possible for the engine to end up in a non-responsive state, halting the fuzzing process. Since unknown problems could arise in the engine and the target is not the Open-TEE engine itself, less effort is spent on discovering and fixing bugs in it. To overcome this, the CA and engine is restarted by AFL after a timeout. This could possibly affect certain test cases on the TA as the engine might become non-responsive before the TA has processed the mutated data. However, such cases are rare and will therefore not have a great impact on the fuzzing. Finally, AFL needs to be able to reproduce a test case by providing the same input. The content of the secure storage used by the PKCS#11 TA to store cryptographic tokens can affect the outcome of the test case. Therefore, after each new fuzzer created test case executed on the target, the PKCS#11 state will be reset by cleaning the secure storage.

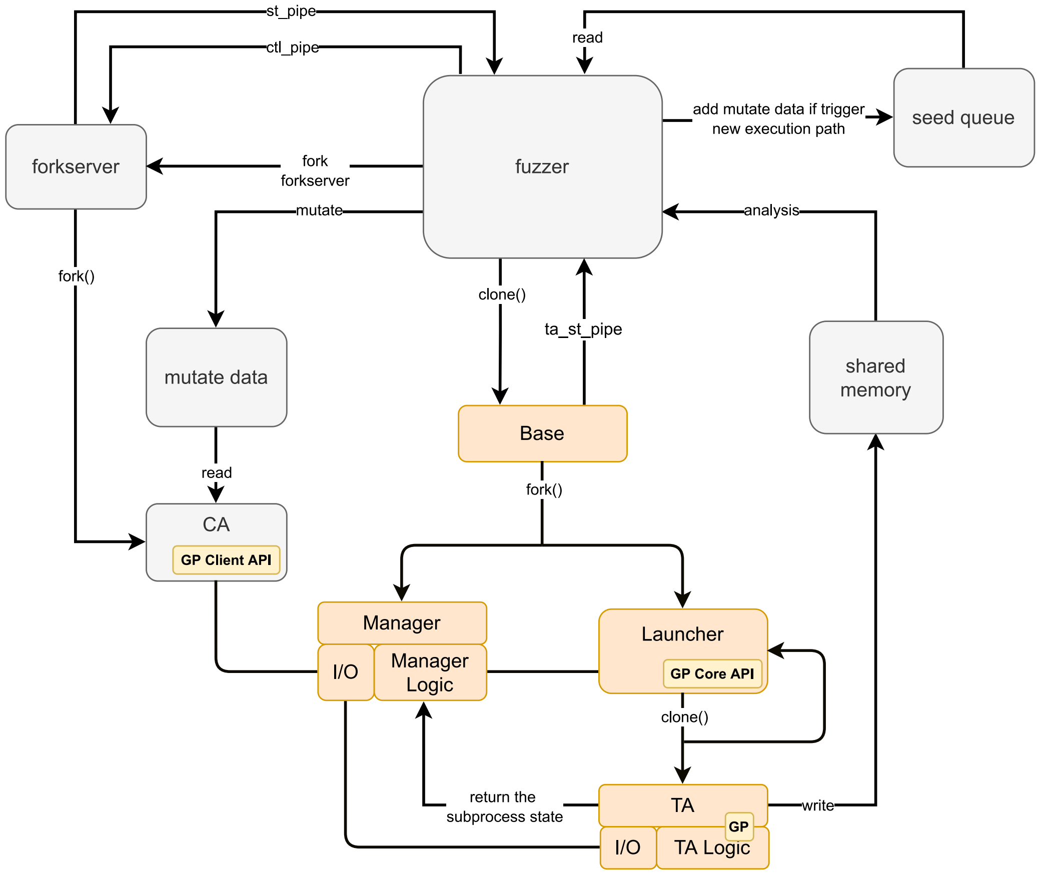

Figure 4.5: Overview of the modified AFL and Open-TEE in the final solution.

4.3 Fuzzing Harness

4.3.1 Fuzzing Entry Points

Now that AFL is modified into working with Open-TEE and able to retain a fuzzing session, a harness will be implemented. The harness will be a CA that handles AFL mutated data and extract the command ID and data that it prepare and send to the target TA.

The PKCS#11 TA takes input normally from a CA via the GP API. This is the most likely input vector that an attacker can take control over which makes it an excellent fuzzing entry point. Another possible input vector to the PKCS#11 TA is via the TA2TA interface. PKCS#11 uses this interface to communicate with an OP-TEE firmware service to attain additional seed material to the PRNG. However, the PKCS#11 TA will connect to this specific TA using its unique UUID. Hence, this approach requires the attacker to have taken control of the OP-TEE provided pseudo trusted application (PTA) itself. This is considered to be outside the scope of this project. Another source of input to the PKCS#11 are the persistent objects stored in OP-TEEs secure storage. All objects are encrypted and integrity protected in the secure storage. Therefore, this approach will be considered unfeasible.

4.3.2 Targeting the PKCS#11 Serialization Parser

PKCS#11 TA offers a variety of commands often requiring multiple parameters, e.g. shared memory for commands arguments and a TA output buffer, and command ID for the TA invocation. For certain command invocations, the CA will perform serialization on the input data transforming the data into a type-length-value format. This data will then be deserialized when it reaches the TA. However, since the serialization is performed in the normal world, an adversary would have control over it and is able to send unserialized data. This allows the fuzzing to be performed with and without serialization (as a binary blob), creating two different fuzzing scenarios. With serialization enabled, the input space will be limited, as the input might be rejected if it is not well-formed during the serialization process before it reaches the TA. Disabling serialization can further exercise the TA handling of unexpected inputs. In the fuzzing sessions the focus will be on command invocations that sends serialized data to the TA, as the deserialization parsing on the TA side is complex and can often be a pitfall for vulnerabilities. This will require building a harness that will support different command invocation that uses serialization.

4.3.3 Harness Design and Data Collection

The harness in the CA is made easily expandable. The CA will read the command ID of the AFL test case and choose the correct setup required for this. This can include creating shared memory allocations, performing initial invocations to set the PKCS#11 TA some state and finally performing the command invocation with the fuzzing data. The setup used will be based on the command ID extracted from the AFL test case.

The baseline provided to AFL has to be a valid input. In order to provide AFL with more learning material, the xtest application in OP-TEE together with its PKCS#11 test suite is ported to Open-TEE. The test cases in xtest will be used for building the AFL corpus. The control buffer provided to TA via the GP API will be recorded during run-time when performing the OP-TEE test suite and dumped to the file system. The dumped data consists of valid serialized data. It is written in binary format together with the command ID it is related to. The binary format is in the form [**command id**]||[**serialized data**]. The input is then read by AFL and split by the harness into two fields which is then passed to the TA via the GP API using shared memory prepared by the harness. AFL is smart enough to learn that command IDs are always of the same length by measuring coverage generated by different mutated inputs. By providing additional initial inputs with the same command ID, but different data, AFL will be able to notice characteristics of the input format with more ease.

4.4 Fuzzing in Iterations

When starting a fuzzing instance on a target with AFL and gcov instrumentation, information about the fuzzing session will be provided to the user. The gcov information from the TA will provide the user with additional information to complement the AFL coverage information. The AFL instrumentation will write to a bitmap telling the fuzzer if any new unique edges has been found and the hit count of each found edge. However, the edges cannot be related back to the original source code. This is possible with the additional coverage information from gcov. It is used with lcov, the graphical interface front-end for gcov, to present the code lines exercised in the target as well the hit count in the form of a navigable website. The results provide information that can help the user to discover problems of the fuzzing process. For example, if the harness or target needs to be adjusted to bypass a certain condition, or if additional learning material needs to be provided to the fuzzer to reach deeper parts of the target. The AFL status screen displays information about the fuzzing instance e.g. the number of unique edges exercised in the target, the current fuzzing strategy used to create new test cases, the number of crashes recorded, lapsed time since last new finding (e.g. edge, crash, timeout) and other statistical information. This information can be used in every new fuzzing iteration to determine how long to run the ongoing fuzzing session, if the fuzzing session needs to be stopped due to crash detection that needs to be patched to not disturb the fuzzing process, or ran until the target is determined to be safe enough.

Result

This section will present the main results of the thesis. Firstly, the core part of the solution will be presented. Secondly, a use case of the solution will be articulated. Finally, the current status of the solution will be unfolded, including the tests performed on it and the evaluation of the outcome; some limitations of the project will be discussed as well.

5.1 TA Fuzzing Tool

The solution is a tool that can be used to fuzz any GP compliant TA that is ported to the Open-TEE environment using the AFL fuzzer. It can provide memory leak, bug detection and other features provided by different sanitizers. The solution also overcomes any possible hangs by using a timer to restart a new test case. Coverage and fuzzing related information is provided to the user to security evaluate the target and debug the fuzzing process. Using this solution also allow the usage of existing tools for mainstream OSes. For example, GNU Debugger (GDB) can be used to perform a root cause analysis upon finding bugs, or simply for troubleshooting purposes. The solution is independent from any TEE specifics and can be used to fuzz TA inside of different TEEs. The tool has currently only been tested on Debian GNU/Linux 10 (Buster) 64-bit, and the modifications are based on AFL++ version ++4.03a (dev), Open-TEE latest release as of 2022 and mbedtls 3.1.0.

5.2 Proof of Concept Fuzzing of PKCS#11 TA

To test the TA fuzzing tool and to demonstrate the usage of it, the PKCS#11 TA in OP-TEE has been used. An example port of the TA Open-TEE has been performed. A simple harness is implemented with the focus on fuzzing the serialization logic inside of the PKCS#11 TA. This will serve as a template for how the harness can be designed. The xtest PKCS#11 test suite is also used to demonstrate how data collection can be performed to build a corpus baseline for AFL. Any existing logging functionalities in the PKCS#11 TA has been modified to write to syslog. This will allow already existing log messages to be displayed during run-time.

5.3 The State of the Solution

Tests have been conducted on the solution using the example PKCS#11 TA port, harness and corpus generated from the xtest test suite. Some tests ran for multiple days without any interruption of the fuzzing process. During these runs, the fuzzer was stably able to detect new edges in the PKCS#11 TA. Deliberate bugs has been added to the ported TA to test ASAN and LSAN. Both programming faults and memory leaks could be detected by AFL. Reports of the coverage information can also be generated after ending a fuzzing session. An intractable interface can be used to browse the source code files, to display the exercised lines and hit count. After running the test suite, on the ported PKCS#11 TA, some tests have failed. Most of them are related to cryptographic operations for specific algorithms, where the output resulted in invalid mechanism error. This indicates that either the porting or the implementation of GP internal core API in Open-TEE can be improved. Another possible reason could be that features are missing in Mbed TLS. This can affect the fuzzing as the fuzzer might not be able to bypass conditions or states that are dependent on these operations. Due to time constraints, the issue has not been further investigated. Certain problems have been found with Open-TEE engine when testing the solution. Any problem causing a major impact on the fuzzing process has been patched. For example, a file descriptor leak is found causing the file descriptors to be used up due to the persistent mode in use. The fuzzing is able to run uninterrupted, however, during longer fuzzing sessions it is possible for the Open-TEE engine to hang and result in a lost test case. The test cases in these hangs are also written to the file system. When testing the test cases from hangs individually they have been executed normally. This indicates that the Open-TEE might transit into a bad state after longer runs. However, this is not a frequent event in the context of fuzzing. A short fuzzing session with an instrumented Open-TEE has also been tested before. A use-after-free error is detected by ASAN, however, no review is performed since Open-TEE is not the fuzzing target and the limited time for the project.

Conclusion

There is currently, to the best of our knowledge, no implementation of source-code-utilizing TA fuzzing. The thesis addresses this gap by providing a proof of concept solution to this problem. The achievements of the thesis are summarized in this section. The section will also consist of ideas of improvements to the project.

6.1 Project Achievements

The goal of the thesis is to investigate how TAs can be fuzzed effectively with the use of source code and test it on the fuzzing target PKCS#11 TA in the OP-TEE project. After analyzing the nature of TAs and the relevant technology, different approaches to the thesis problem are proposed. The benefits and challenges of multiple proposed fuzzing environment are discussed. Additionally, some available fuzzing techniques, tools and features available are motivated and used to achieve more efficient fuzzing. Then a proof of concept is implemented to demonstrate and review the effectiveness of the solution. This includes TA hosting, source code powered fuzzing, techniques and features that can improve the fuzzing efficiency, monitoring capabilities for bug detection, extended debugging information, an example porting of the fuzzing target from its native environment to the fuzzing environment, fuzzing harness for the fuzzing target to showcase how the solution can be used. The baseline for the target is also formed by extracting data from an existing PKCS#11 TA test suite, creating a corpus in the data format that is compatible with fuzzing harness. All these combined allows for a flexible TA fuzzing solution that is not bound to TEE specific TAs, and can easily be extended with additional tools and features.

6.2 Future work

Some possible improvements to known problems in the project and additional new features are discussed here.

6.2.1 Optimization

In the current state of the project, existing bugs in the Open-TEE engine can hang and trigger a timeout procedure, after a user specified time. The time impact relatively insignificant as it happens infrequently. However, patching the root of existing bugs will allow smoother runs. The fuzzing process is currently only utilizing a single core. AFL supports multi core utilization, however Open-TEE is designed to only allow a single instance of the engine. For now, it's only possible to run multiple instances of the project in isolated environments. A rework in Open-TEE will allow more CPU utilization.

6.2.2 Generalization

Open-TEE is currently missing features offered by other TEEs such as crypto features, GP API extensions, implementation of PTAs. If the project can extend and cover these features of targeted TEEs a generalized TEE port can be achieved resulting in more seamless porting of custom TAs.

6.2.3 Coverage

The fuzzing in the project can be improved using white-box fuzzing techniques. For example, concolic fuzzing uses symbolic execution and constraint solvers to complement the coverage-guided fuzzing [64]. Fuzzing is fast and is good at finding test cases that solves loose edge conditions. On the contrary, symbolic execution is slow and good at discovering test cases that solves complex edges with tight conditions. This will allow better coverage in the expense of performance. The provided harness serves as a reference, but can be improved by expanding the type of TA invocations it covers. The harness also only focuses on exercising the code related to serialization logic. However, evaluation on other parts of the PKCS#11 TA are of interest as well from a security perspective. Improving the corpus will also have a great impact on the discovery of new paths. For example, additional PKCS#11 example data can be gathered and the current data can be optimized by removing similar test cases and test cases that are intended to fail. This will provide AFL with a better baseline, speeding up the coverage discovery.

Bibliography

[1] E. Carriere. "Pkcs11," Linaro. (), [Online]. Available: `https://github.com/OP-TEE/optee_os/tree/master/ta/pkcs11`. (accessed: 9.14.2022).

[2] M. Alenezi and M. Zarour, "On the relationship between software complexity and security," *arXiv preprint arXiv:2002.07135*, 2020.

[3] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek, "{Hardware-assisted}{on-demand} hypervisor activation for efficient security critical code execution on mobile devices," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 565–578.

[4] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, IEEE, vol. 1, 2015, pp. 57–64.

[5] *Pkcs #11 cryptographic token interface base specification version 2.40*, English, version 2.40, OASIS, 2015, 149 pp. [Online]. Available: `https://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.pdf`.

[6] *Op-tee documentation*, English, version 3.9.0, Trusted Firmware, 2020, 165 pp. [Online]. Available: `https://optee.readthedocs.io/_/downloads/en/3.9.0/pdf/`.

[7] C. Carabas and M. Carabas, "Fuzzing the linux kernel," in *2017 Computing Conference*, IEEE, 2017, pp. 839–843.

[8] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft.," *Queue*, vol. 10, no. 1, pp. 20–27, Jan. 2012, ISSN: 1542-7730. DOI: `10.1145/2090147.2094081`. [Online]. Available: `https://doi.org/10.1145/2090147.2094081`.

[9] H. Yang and M. Lee, "Demystifying arm trustzone tee client api using op-tee," in *The 9th International Conference on Smart Media and Applications*, 2020, pp. 325–328.

[10] *Security in an armv8 system*, English, version 1.0, Arm, 2017, 12 pp. [Online]. Available: `https://developer.arm.com/documentation/100935/0100/Security-in-ARMv8-A-systems-`.

[11] *Cortex-a8 technical reference manual*, English, version r1p1, Arm, 2006, 730 pp. [Online]. Available: `https://developer.arm.com/documentation/ddi0344/b?lang=en`.

[12] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: Enforcing kernel code integrity on the trustzone architecture," *arXiv preprint arXiv:1410.7747*, 2014. [Online]. Available: `https://arxiv.org/ftp/arxiv/papers/1410/1410.7747.pdf`.

[13] *Arm security technology building a secure system using trustzone® technology*, English, Arm, 2005, 108 pp. [Online]. Available: `https://developer.arm.com/documentation/PRD29-GENC-009492/c/preface?lang=en`.

[14] "Trusted board boot," Arm. (), [Online]. Available: `https://github.com/ARM-software/arm-trusted-firmware/blob/master/docs/design/trusted-board-boot.rst`. (accessed: 25.02.2022).

[15] "What is root of trust?" Thales. (), [Online]. Available: `https://cpl.thalesgroup.com/faq/hardware-security-modules/what-root-trust`. (accessed: 24.02.2022).

[16] "About," Trusted Firmware. (), [Online]. Available: `https://www.trustedfirmware.org/about/`. (accessed: 13.03.2022).

[17] "Technical overview of trusted firmware-a embedded linux conference," Arm. (), [Online]. Available: `https://elinux.org/images/0/05/Elc-tfa.pdf`. (accessed: 15.06.2022).

[18] "Op-tee, open-source security for the mass-market," Linaro. (), [Online]. Available: `https://www.linaro.org/blog/op-tee-open-source-security-mass-market`. (accessed: 25.02.2022).

[19] *Trusted firmware-a*, English, version 2.6, Trusted Firmware, 2021, 632 pp. [Online]. Available: `https://trustedfirmware-a.readthedocs.io/_/downloads/en/v2.6/pdf/`.

[20] "Open portable trusted execution environment," Linaro. (), [Online]. Available: `https://www.op-tee.org/`. (accessed: 22.02.2022).

[21] "Op-tee overview," STMicroelectronics. (), [Online]. Available: `https://wiki.st.com/stm32mpu/wiki/OP-TEE_overview#cite_note-optee.org-1`. (accessed: 25.02.2022).

[22] "Tee subsystem," The kernel development community. (), [Online]. Available: `https://www.kernel.org/doc/html/latest/staging/tee.html`. (accessed: 25.02.2022).

[23] "Secure storage," Trusted Firmware. (), [Online]. Available: `https://optee.readthedocs.io/en/latest/architecture/secure_storage.html`. (accessed: 25.02.2022).

[24] B. McGillion, T. Dettenborn, T. Nyman, and N. Asokan, "Open-TEE – an open virtual trusted execution environment," Aalto University, Tech. Rep., 2015. eprint: `{\ttarXiv:1506.07367[cs.CR]}`. [Online]. Available: `http://arxiv.org/abs/1506.07367`.

[25] J. E. Brian McGillion *et al.* "Open-tee." (), [Online]. Available: `https://open-tee.github.io/faq/`. (accessed: 21.06.2022).

[26] "Trustcom2015 open-tee," Intel. (), [Online]. Available: `https://github.com/Open-TEE/Open-Tee.github.io/raw/master/documents/TrustCom2015_OpenTEE.odp`. (accessed: 21.06.2022).

[27] *Tee client api specification*, English, version 1.0, GlobalPlatform, 2010, 58 pp. [Online]. Available: `https://globalplatform.org/wp-content/uploads/2010/07/TEE_Client_API_Specification-V1.0.pdf`.

[28] *Tee system architecture*, English, version 1.1, GlobalPlatform, 2017, 43 pp. [Online]. Available: `https://globalplatform.org/wp-content/uploads/2017/01/GPD_TEE_SystemArch_v1.1_Public_Release.pdf`.

[29] J. Li, B. Zhao, and C. Zhang, "Fuzzing: A survey," *Cybersecurity*, 2018. DOI: `https://doi.org/10.1186/s42400-018-0002-y`.

[30] J. Reimer. "5 cves found with feedback-based fuzzing." (), [Online]. Available: `https://www.code-intelligence.com/blog/5-cves-found-with-feedback-based-fuzzing`. (accessed: 16.11.2021).

[31] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[32] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1980–1997, 2019.

[33] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[34] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

[35] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, USENIX Association, Aug. 2020.

[Online]. Available: `https://www.usenix.org/conference/woot20/presentation/fioraldi`.

[36] h. vanhauser-thc hexcoder- *et al.* "Aflplusplus." (), [Online]. Available: `https://github.com/AFLplusplus/AFLplusplus`. (accessed: 19.06.2022).

[37] "Aflplusplus," AFL community. (), [Online]. Available: `https://aflplus.plus/`. (accessed: 19.06.2022).

[38] R. Fan, J. Pan, and S. Huang, "Arm-afl: Coverage-guided fuzzing framework for arm-based iot devices," in *International Conference on Applied Cryptography and Network Security*, Springer, 2020, pp. 239–254.

[39] M. Zalewski, *Afl documentation*, English, version 2.53b, 2019, 87 pp. [Online]. Available: `https://afl-1.readthedocs.io/_/downloads/en/latest/pdf/`.

[40] ——, "Binary fuzzing strategies: What works, what doesn't." (), [Online]. Available: `https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html`. (accessed: 19.06.2022).

[41] ——, "Fuzzing random programs without execve()." (), [Online]. Available: `https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html`. (accessed: 14.03.2022).

[42] l. vanhauser-thc llzmb *et al.* "Llvm_mode persistent mode." (), [Online]. Available: `https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.persistent_mode.md`. (accessed: 14.03.2022).

[43] M. Rash. "Afl-cov - afl fuzzing code coverage." (), [Online]. Available: `https://github.com/mrash/afl-cov`. (accessed: 20.06.2022).

[44] "Introduction to gcov," Free Software Foundation. (), [Online]. Available: `https://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html#Gcov-Intro`. (accessed: 20.06.2022).

[45] oberpar. "Readme file for the ltp gcov extension (lcov)." (), [Online]. Available: `https://github.com/linux-test-project/lcov`. (accessed: 20.06.2022).

[46] "Addresssanitizer," Google. (), [Online]. Available: `https://github.com/google/sanitizers/wiki/AddressSanitizer`. (accessed: 14.03.2022).

[47] "Addresssanitizer," Google. (), [Online]. Available: `https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer`. (accessed: 14.03.2022).

[48] "Memorysanitizer," Google. (), [Online]. Available: `https://github.com/google/sanitizers/wiki/MemorySanitizer`. (accessed: 14.03.2022).

[49] "Undefinedbehaviorsanitizer," Clang. (), [Online]. Available: `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`. (accessed: 14.03.2022).

[50] "An introduction to pkcs#11," Thales Group. (), [Online]. Available: `https://thalesdocs.com/gphsm/ptk/5.9/docs/Content/PTK-C_Program/intro_PKCS11.htm`. (accessed: 14.03.2022).

[51] M. Busch, A. Machiry, C. Spensky, G. Vigna, C. Kruegel, and M. Payer, "Teezz: Fuzzing trusted applications on cots android devices," in *2023 IEEE Symposium on Security and Privacy (SP)*, IEEE Computer Society, 2022, pp. 220–235.

[52] M. Ghasemisharif, "State of the fuzz: An analysis of black-box vulnerability testing," [Online]. Available: `https://github.com/galli-leo/emmutaler/blob/master/docs/thesis.pdf`.

[53] S. Makkaveev. "The road to qualcomm trustzone apps fuzzing." (), [Online]. Available: `https://research.checkpoint.com/2019/the-road-to-qualcomm-trustzone-apps-fuzzing/`. (accessed: 20.08.2022).

[54] J. Fenlason and R. Stallman, "Gnu gprof," *GNU Binutils. Available online: http://www.gnu.org/software/binutils (accessed on 4 October 2022)*, 1988.

[55] J. E. Brian McGillion *et al.* "Open-tee." (), [Online]. Available: `https://github.com/Open-TEE`. (accessed: 21.06.2022).

[56] C. Carabas and M. Carabas, "Fuzzing the linux kernel," in *2017 Computing Conference*, 2017, pp. 839–843. DOI: `10.1109/SAI.2017.8252193`.

[57] S. Schumilo, R. Spenneberg, and H. Schwartke, "Don't trust your usb! how to find bugs in usb device drivers," *Blackhat Europe*, 2014.

[58] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018, Montpellier, France: Association for Computing Machinery, 2018, pp. 475–485, ISBN: 9781450359375. DOI: `10.1145/3238147.3238176`. [Online]. Available: `https://doi.org/10.1145/3238147.3238176`.

[59] R. Swiecki. "Honggfuzz," Google. (), [Online]. Available: `http://code.google.com/p/honggfuzz`. (accessed: 10.4.2022).

[60] “Libfuzzer – a library for coverage-guided fuzz testing.,” LLVM Project. (), [Online]. Available: `https://llvm.org/docs/LibFuzzer.html`. (accessed: 10.4.2022).

[61] C. D. Sylvestre Ledru. “Peach,” Mozilla. (), [Online]. Available: `https://github.com/MozillaSecurity/peach`. (accessed: 10.4.2022).

[62] A. Hazimeh, A. Herrera, and M. Payer, “Magma: A ground-truth fuzzing benchmark,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 1–29, 2020. [Online]. Available: `https://hexhive.epfl.ch/publications/files/21SIGMETRICS.pdf`.

[63] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, “Fuzzbench: An open fuzzer benchmarking platform and service,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1393–1403.

[64] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, “{Qsym}: A practical concolic execution engine tailored for hybrid fuzzing,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 745–761.

Printed by Tryckeriet i E-huset, Lund 2022

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2022-905
http://www.eit.lth.se