
**Budapest University of Technology and Economics**
Faculty of Electrical Engineering and Informatics
Department of Networked Systems and Services

# Implementing Cryptographic Operations Using OP-TEE on Embedded Platforms

BSc Thesis

*Author*
Gábor Székely

*Supervisor*
Dr. Levente Buttyán

December 6, 2018

# Contents

# Abstract

In the last couple of years, the Internet of Things (IoT) has become very popular. IoT refers to everyday objects that have been in some way connected to the Internet, in order to offer new capabilities, such as: data collection, remote control through a webapp and so on. To make this possible, embedded computers are placed in these devices, which means, that these computers often have limited resources. Limited resources have resulted in sub par security in many IoT devices, which is a major problem, since the data and functions these devices operate on are often sensitive or critical.

In order to protect the data handled by the device, cryptographic operations can be utilized. In this paper, I examine mechanisms that make use of cryptographic operations such as remote access and secure communications. A recurring requirement (among others) in these functions is the need for protecting the cryptographic keys that are in use, preferably even if the device was physically compromised. Considering the amount of different tasks, that require the same properties, a common interface would eliminate the need for reimplementing the same functions for all the applications. For this I chose the PKCS#11 API.

To protect the cryptographic keys, some specialized hardware support is necessary. Usually cryptographic co-processors or security co-processors are added to the board to fulfill this need. However, these often increase the cost of the device, and one of the reasons for the success of IoT is its very low price. In order to increase the likelyhood of adoption by not needing to place extra hardware elements on the board, I decided to utilize ARM's TrustZone technology.

# Kivonat

Az elmúlt néhány évben a Dolgok Internete (angolul Internet of Things, innen az IoT rövidítés) nagy népszerűségre tett szert. Az IoT lényegében a mindennapi dolgaink internethez csatlakoztatását foglalja magában. Ennek különböző előnyei lehetnek, mint például a távoli adatgyűjtés lehetővé tétele, vagy az IoT eszköz távoli irányítása egy webalkalmazás segítségével. Az internettel való kapcsolatot, illetve az új funkciókat egy beágyazott számítógép teszi lehetővé az eszközben. A beágyazott számítógépek természetüknél fogva limitált erőforrásokkal rendelkeznek. Ez azonban gyakran átlagon aluli biztonságot eredményez az IoT eszközökben, ami egy jelentős probléma, mivel ezek az eszközök sok esetben kritikus, illetve érzékeny adatokkal dolgoznak.

Annak érdekében, hogy megvédjük az adatokat, amivel az eszközünk dolgozik, kriptográfiai műveleteket használhatunk. Dolgozatomban megvizsgálom a különböző mechanizmusokat amik kriptográfiai műveleteket használnak, mint például a távoli hozzáférés és biztonságos kommunikáció. Többek között, egy visszatérő követelmény ezekben a funkciókban a kriptográfiai kulcsok védelme, lehetőleg még abban az esetben is, ha maga az eszköz fizikailag kompromittálódik. Mivel számos különböző feladatnak ugyanazokra a tulajdonságokra van szüksége, egy közös interfész előnyös lenne, hogy ne kelljen újra implementálni ugyanazokat a funkciókat a különböző alkalmazásokban. Erre a PKCS#11 API-t választottam.

A titkos kulcsok védelméhez specializált hardver támogatásra van szükség. Általában emiatt kriptográfiai koprocesszorokat adnak az eszközhöz. Azonban ezek megnövelik az eszköz árát, és az egyik fő oka az IoT sikerének az eszközök alacsony ára. Ezért az elterjedés könnyítésének érdekében olyan technológiát kerestem, mely nem igényel további hardver elemeket és az ARM TrustZone mellett döntöttem.

# Chapter 1

# Introduction

Moore's law [13] – a prediction that the number of transistors on a single silicon die doubles around every 2 years – has been holding for more than 50 years. Combining this steady increase in computing power and shrinking size with decreasing price resulted in small computers that can be embedded in many everyday items, such as fridges, weather measurement stations, home thermostats, cars, coffee makers, yoga mattresses, lightbulbs, doorbells, *etc.* Moreover, as internet access became cheaper and widely accessible, these embedded computers are now connected to the Internet, to enable remote operation and monitoring, as well as to allow embedded devices to perform network based transactions like shopping or uploading data in the cloud. The resulting ecosystem has come to be called the *Internet of Things* or IoT for short, and the devices that belong to this ecosystem are called *IoT devices* or "smart" devices.

However, besides the advantages that Internet connection can bring, a new attack surface for adversaries is created, which can expose devices that were before unreachable for attackers. The impact of attacks on smart medical devices, connected vehicles or smart factories is high, potentially leading to significant monetary loss, or even danger to human life. On the other hand, as a result of the history of IoT, low price remains one of the main objectives, and when costs need to be saved in a product, security is usually one of the first aspects that suffers. It is also very hard for the average consumer to evaluate the security of a device, so there is no monetary incentive for investing a lot of money into producing a very secure product until a huge cyber incident happens. This has led to IoT devices on the market having terrible security: default passwords in realeased devices [11], and many incidents, like the gigantic DDoS attack on major web based services by the Mirai botnet that is built from compromised IoT devices [2].

In this work, I aim to improve the security of communication in the Internet of Things. Embedded IoT devices should be able to communicate securely with their owner or operator. This is needed not only for data transmission but also for providing secure remote access to the device for configuration and management purposes. Secure communications can be implemented with cryptographic mechanisms. However, long-term secret passwords

or private keys needed for authenticating the device to the owner/operator when setting up a secure communication session should not appear in memory accessible to the potentially compromised firmware/OS. This means that such secret keys must be stored in secure storage and the cryptographic operations that use them should be implemented by trusted applications executed in a so called Trusted Execution Environment. There are a number of ways to establish such a Trusted Execution Environment and secure storage, for example some ARM processors supply the necessary tools to do this.

The idea to implement a keystore in the TEE is not new. Android has the Android Keystore [6], which is accessible through the KeyChain API and the Android Keystore provider. If the manufacturer of the phone provides a driver that enables it to use a TEE, it will do so, otherwise it defaults to a software implementation. Unfortunately, these focus on smartphones so they are not suitable for my goals.

This approach presents a new problem: since the secrets are in the Trusted Execution Environment, legitimate programs cannot access them either. We need to create a way for these programs to request operations with the secrets stored in the Trusted Execution Environment, but make it impossible to directly access them – since if the legitimate programs could do that, then an adversary in a compromised firmware/OS could steal these secrets. A uniform API fulfills this requirement, while allowing us to make this functionality available for multiple purposes. There exist standard cryptographic APIs for this, such as the PKCS#11 API, which is a widely used standard that many client applications already support. However, PKCS#11 is a rather large specification and in the past a number of vulnerabilities have been found in the API [3, 5]. A possible way to try to avoid being affected by vulnerabilities that may be discovered in the future is by using only a subset of the interface, since the standard allows this. This can be observed in practice in [4], where a provably secure cryptographic interface is constructed and an emulation of this security policy is created by restricting the PKCS#11 interface.

Another example of an API that allows operations on stored keys and offers protection for them can be found in cryptographic tokens such as smart cards and hardware security modules (HSM) is the IBM CCA (Common Cryptographic Architecture). IBM CCA is primarily used by banks (for example in ATMs, with IBM cryptoprocessors providing the interface. [4]

The rest of the paper is organized in the following manner: In Section 2, I introduce the design of my proposed solution. In Section 3, I expand on the details of my implementation. In Section 4, I evaluate my results, and finally, in Section 5, I give a summary of my work and some possible future improvements.

# Chapter 2

# Design

## 2.1 Used System Components

### 2.1.1 ARM TrustZone

As already mentioned, I wanted to achieve some kind of separation between the environment where the standard operating system, i.e. Linux and the various client applications reside – from here on REE (Rich Execution Environment) or NW (Normal World) – and the environment where the cryptographic secrets are stored and are operated on. This environment will be called TEE (Trusted Execution Environment) or SW (Secure World). The ARM TrustZone technology makes this possible by enforcing system-wide hardware isolation. Different resources of the Sytem on Chip (SoC) can be placed in either subsystem, and NW assets will be unable to control SW assets, while SW assets can access anything. This is enforced using an extra bit in the addressing called NS Bit [1]. For example, it is possible to segment the working memory of the device this way, so even the Linux kernel can't read or write the areas that belong to SW. Figure 2.1 showcases the high level overview of the hardware isolation.

### 2.1.2 GPD TEE Specification

The GPD TEE Specification defines the requirements a TEE has to fulfill and the various interfaces that are used to connect the components of a system with a TEE. This specification is not meant only for ARM TrustZone, but as a general way to create a Trusted Execution Environment, if the necessary hardware is available.

The high level requirements for a TEE are the following: Code that executes in the TEE must be authenticated, while the confidentiality and integrity of TEE assets and data needs to be protected. Both local and remote attacks, and even some set of hardware attacks

[1] `https://www.timesys.com/security/trusted-software-development-op-tee/` (last visited on 2018–10–23)

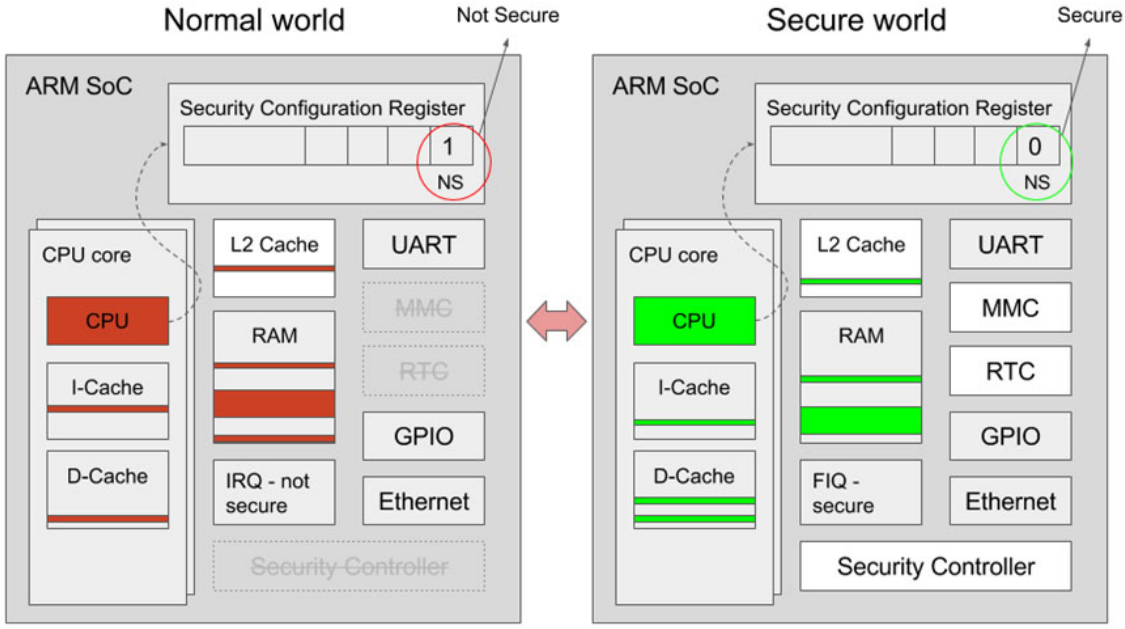


**Figure 2.1.** *Hardware overview example of ARM TrustZone Security*[1]

need to be resisted by the TEE. Unauthorized tracing, control and test features have to be impossible. [7]

In the GPD TEE architecture, there is a TEE OS, the operating system running in the TEE and providing internal APIs, separation for Trusted Applications, which are like userland applications in Linux. Trusted storage is a kind of persistent storage that is only available in the TEE, protected either by hardware or cryptographic methods [9]. Figure 2.2 outlines the software architecture defined by the GPD TEE Specification.

The GPD TEE Internal Core API contains a Cryptographic Operations API [8], that provides an interface to perform cryptographic operations with the help of the TEE. Depending on the actual TEE and its configuration, the algorithms may be implemented in software or backed by hardware as well. However, as long as the given TEE follows the standard we can be sure that the API calls do what they are specified to do and the data and keys can't be accessed from the Normal World. These properties would make this interface ideal for our use case, however this API is only exposed in Secure World, and also not as widely used in client applications as PKCS#11 for instance.

### 2.1.3 OP-TEE

There are numerous TEE implementations, but it was still hard to find one that fits our needs. One of the reasons is that many of the TEEs are closed source, and developing applications for them is either impossible, or requires signing NDAs. It is also reasonable to limit our scope to TEEs that are compatible with ARM chips, because IoT devices are mostly based on such SoCs. As a result I chose to rely on TrustZone technology. [16] places TEEs based on TrustZone technology into two groups: industrial and academic TEEs. I

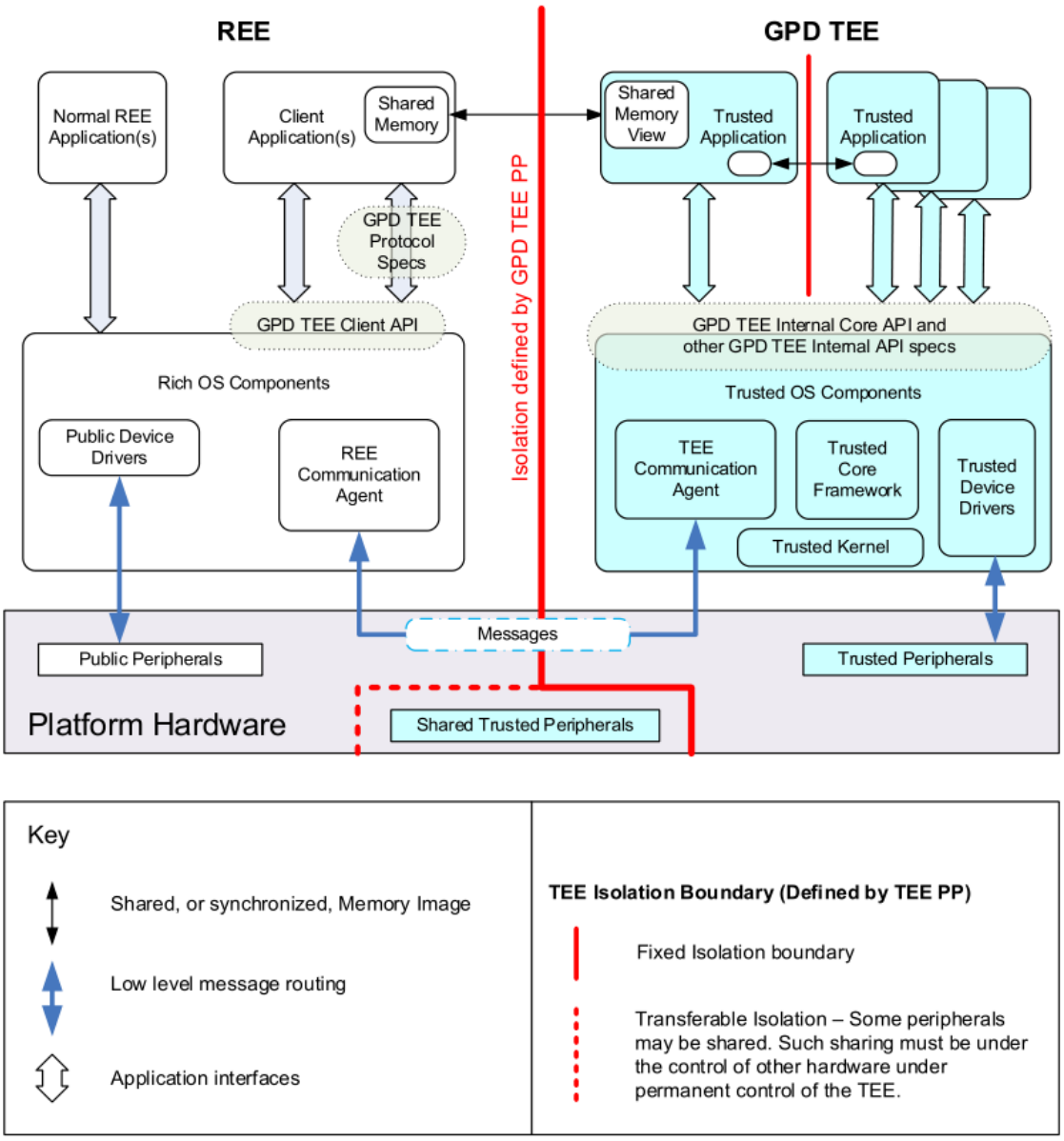


**Figure 2.2.** *TEE software architecture (source: [9])*

wanted my solution to be usable in real world devices, so I chose an industrial TEE. Some of the industrial TEEs mentioned in [16]:

- QSEE
- Trustonic
- Securi-TEE
- SierraTEE
- OP-TEE
- Nvidia TLK

Out of these, closed source options were instantly discarded, since they don't allow modification of the system, and that capability may be necessary for us. From the remaining TEEs I chose OP-TEE, as it seemed to be the easiest to access – it is available on GitHub – and had the most active development. It also supports a wide range of devices.

Another option would be using a separation kernel that is mathematically proven to correctly separate different running processes. An example of this is the seL4 microkernel [12]. However the drawback of this aproach is that this limits adoption, since in my experience there aren't as many programs written for these microkernels and there are fewer people who would have the knowledge to create applications in such an environment.

"OP-TEE is an open source project which contains a full implementation to make up a complete Trusted Execution Environment. The project has roots in a proprietary solution, initially created by ST-Ericsson and then owned and maintained by STMicroelectronics. In 2014, Linaro started working with STMicroelectronics to transform the proprietary TEE solution into an open source TEE solution instead. In September 2015, the ownership was transferred to Linaro. Today it is one of the key security projects in Linaro, with several of Linaro's members supporting and using it."[2]

The source of the OP-TEE project is available on GitHub, in three separate repositories: optee_os[3] holds the OP-TEE Kernel, optee_client[4] holds the NW code supplying the OP-TEE Client API and optee_test[5] holds a regression test suite. OP-TEE follows the GPD TEE Specification[6].

### 2.1.4 Linux

Linux is a very common operating system in IoT, most of the devices where real time response is not required and have the necessary resources use Linux. All software components are provided to use Linux as the REE with OP-TEE (and no other REE at the moment), so I used Linux as the REE OS in my work.

Shared libraries are used in the design later, so I will introduce them here. Shared libraries allow programs to reuse code: a program can load a shared library that will have the implementation of a number of functions that can be used by the program. This makes it possible to transparently switch implementations of certain methods in a program that is made by some other party, by switching the shared library it uses.

[2] `https://www.op-tee.org/about/` (last visited: 2018-10-22)
[3] `https://github.com/OP-TEE/optee_os` (last visited: 2018-10-22)
[4] `https://github.com/OP-TEE/optee_client` (last visited: 2018-10-22)
[5] `https://github.com/OP-TEE/optee_client` (last visited: 2018-10-22)
[6] `thttps://www.op-tee.org/documentation/` (last visited: 2018-10-22)

## 2.2 Cryptographic building blocks

### 2.2.1 PKCS#11

PKCS#11 was originally one of the Public Key Cryptography Standards, that are a set of standards created by RSA Laboratories, but in 2013, further development has been turned over to the OASIS PKCS 11 Technical Committee[7]. PKCS#11, also referred to as Cryptoki, is a general purpose API for accessing devices capable of storing cryptographic keys and executing cryptographic operations [15]. Some examples of such devices are smartcard security tokens, hardware authentication tokens (e.g., Yubikey) and cryptographic hardware security modules (HSMs). PKCS#11 is widely used in the industry, and as a result, a lot of applications support it, including, but not limited to OpenSSL[8], OpenSSH[9], Mozilla Firefox and Mozilla Thunderbird through Mozilla NSS[10], OpenDNSSEC[11]. There are also several tools that provide a user interface to manipulate tokens (e.g.: generate, import, export keys, encrypt or decrypt data), for example the pkcs11-tool that is part of the OpenSC project[12]. It is important to note, that a device may only support a subset of the mechanisms defined in the standard and it would be still compliant, so some tokens may not work with all of these applications, depending on what API calls they support.

### 2.2.2 Relevant security guarantees

A general model of the PKCS#11 system can be seen on Figure 2.3. In the PKCS#11 terminology a token is the device that stores the cryptographic keys, certificates, data, etc. The data that is stored on the token is organized into objects (e.g., a private key object) and can be accessed through handles, that can be likened to the pointers we use in programming. A slot is what we can access a token through (e.g., a smartcard reader). Certain objects can only be used if the client logs in to the token, by providing a PIN. Regardless of whether a user is logged in, if an object has the sensitive flag set, it shall not be revealed off the token, so for example it can't be viewed with the `C_GetAttributeValue` mechanism, but it may be exportable if wrapped with another key. If an object has the unexportable flag set it shall not be exportable at all, and the unexportable flag shall not be modifiable either. These guarantees have to hold, even if the token connected to a slot in a malicious machine. [15]

[7] `https://www.oasis-open.org/news/pr/oasis-enhances-popular-public-key-cryptography-standard-pkcs-11-fo` (last visited 2018. 10. 21.)

[8] `https://developers.yubico.com/YubiHSM2/Usage_Guides/OpenSSL_with_pkcs11_engine.html` (last visited 2018. 10. 21.)

[9] `https://github.com/OpenSC/OpenSC/wiki/OpenSSH-and-smart-cards-PKCS%2311` (last visited 2018. 10. 21.)

[10] `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/PKCS11/Module_Installation` (last visited 2018. 10. 21.)

[11] `https://www.opendnssec.org/softhsm/` (last visited 2018. 10. 21.)

[12] `https://github.com/OpenSC/OpenSC/wiki` (last visited 2018. 10. 22.)

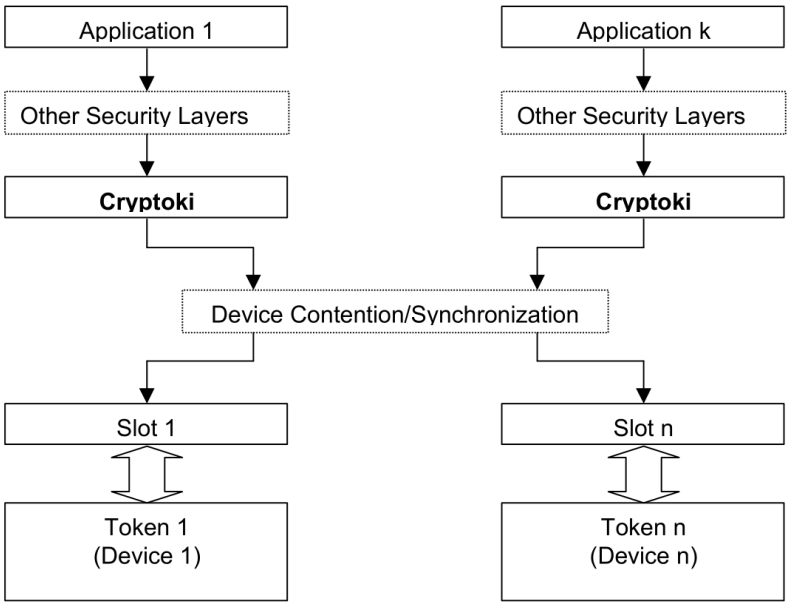


**Figure 2.3.** *General PKCS#11 model (source: [14])*

### 2.2.3 Usage example

Usually the client applications access the PKCS#11 API through shared objects, or dynamically linked libraries based on the operating system. For example, if someone wants to use their smartcard to authenticate to a server with OpenSSH, they would have to find the shared object that is able to communicate with their smartcard. The manufacturer should provide this, or the OpenSC project could be a good place to start looking. Once they have the shared object it can be specified with the `-I` command line flag. Internally OpenSSH includes the `pkcs11.h` header, and calls the functions declared in it. When we specify the shared object, it dynamically links the actual hardware specific implementation for the exact card they want to use.

### 2.2.4 Soft tokens

It is not necessary for the token to be implemeted as a hardware component, it can also be implemented completely in software. Of course this way the security guarantees can't be enforced, but it is useful for developing and testing applications. These were also helpful for me as reference implementations while writing code for this project. First, SoftHSM[13], which is developed as part of the OpenDNSSEC project. OpenDNSSEC uses the PKCS#11 API to handle and store its cryptographic keys. The purpose of SoftHSM is to allow OpenDNSSEC to be used if the user can't afford, or simply doesn't want to use a hardware token. Another example of a soft token is part of the openCryptoki project[14], and it is supposed to be used for testing.

[13] `https://www.opendnssec.org/softhsm/` (last visited 2018. 10. 21.)
[14] `https://github.com/opencryptoki/opencryptoki` (last visited 2018. 10. 21.)

### 2.2.5 SKS

Secure Key Storage (SKS) is a proposal[15] by Etienne Carrière for a token that is implemented as an OP-TEE TA, and accessible through the PKCS#11 API. It is currently under development, when I first started working with it, it only supported AES mechanisms, at the time of writing there is also partial support for RSA and EC mechanisms. The project consists of three parts: the TA implementation of the token, the shared object that provides access to the token from normal world and the regression tests that are integrated with the OP-TEE xtest test suite. The project can be found on GitHub under the user etienne-lms in the `sks` branches of the forks of the OP-TEE repositories.

When there is a PKCS#11 call, the SKS shared object converts the constants from the PKCS#11 values to the ones used by SKS internally and serializes the data, so it can be passed through the OP-TEE TA call interface. In the TA, the data is deserialized, and the requested operations are executed. Most of the time the PKCS#11 mechanisms can be easily translated into GPD TEE Core API calls, so the TA heavily relies on these calls.

The regression tests are based on the tests for the OP-TEE Internal Core API cryptographic methods. These are implemented through a test TA that basically just forwards the calls to the OP-TEE kernel and returns the results. For the SKS tests these calls are not called through the test TA, but SKS. There are also functionalities in SKS – for example `C_Login` – that cannot be tested this way, these have their own tests too.

## 2.3 Architecture

Building on the above described components, we can build a system, where cryptographic keys are strongly protected. The TrustZone technology provides two separated execution environments that can only communicate through well defined interfaces. The lower priviliged environment, called the Rich Execution Environment (REE) or Normal World can house a conventional operating system – in our case Linux – and the higher priviliged one, called Trusted Execution Environment (TEE) or Secure World, holds a trusted operating system, in our case OP-TEE. Inside OP-TEE, we have the SKS, utilizing the separation between the two worlds, storing cryptographic keys in OP-TEE's secure storage and providing the PKCS#11 interface to Normal World. Using the PKCS#11 API we can use the keys stored in SKS by any client application that supports the interface, while the client application can never directly access the key itself. Even in a scenario, where a Linux application with root priviliges is compromised and it can recover the PIN for the SKS token (e.g. by replacing the shared object with a malicous implementation, that reveals the PIN), the attacker can only use the keys stored on it, but not recover and steal them.

In this paper, I chose OpenSSH as the client application. This allows us to copy files with `scp`, connect to a server and run scripts, or even provide a remote shell with ssh reverse

[15] `http://connect.linaro.org.s3.amazonaws.com/hkg18/presentations/hkg18-402.pdf` (last visited 2018. 10. 22.)

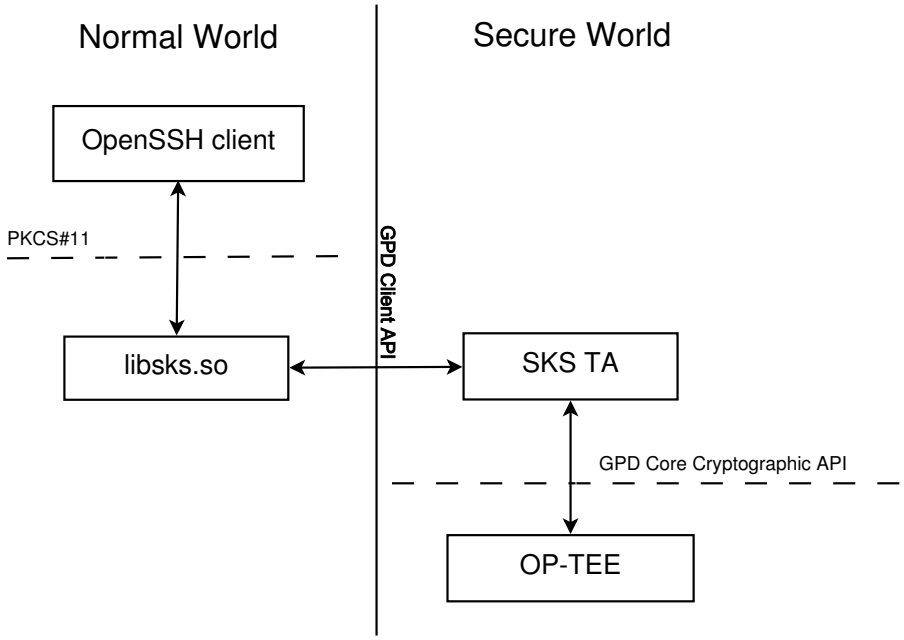


**Figure 2.4.** *SKS communicating with OpenSSH*

tunneling (the reverse tunneling is necessary, because the `-I` option is only available in client mode). OpenSSH is also a nice choice, because it is easy to set up compared to a VPN and can be easily tested by calling `ssh root@localhost`. An overview of this architecture in action can be seen on Figure 2.4

# Chapter 3

# Implementation

## 3.1 Environment

### 3.1.1 QEMU

QEMU[1] (Quick EMUlator) is a machine emulator, it can emulate a target system and run unmodified operating system and programs designed for it, regardless of the host system's CPU architecture and operating system. QEMU can be run on all the major operating systems (i.e., Linux, Windows, Mac OS). In my case, using QEMU came with multiple advantages. The first and obvious one is that this eliminates the need to have access to hardware for development. Other than that, the development process is also faster and more convenient this way: the whole project can be built by issuing one make command, no need to flash SD Cards, or be near a physical device at all. Opening the two different consoles (to Normal and Secure World) is as easy as creating two `screen` sessions. An example of running one of the regression test can be seen on Figure 3.1 and Figure 3.2, which were originally on one desktop, but for better visibility I separated them into two figures. Figure 3.1 has the Normal World console, while on Figure 3.2 the upper window one has the Secure World console and the bottom window was used to build the whole system and has the QEMU console.

### 3.1.2 Build System

#### repo

Repo[2] is a repository management tool originally built for managing the Google AOSP project. It allows collecting the necessary code from multiple git repositories by specifying them in a manifest file. Multiple manifest files can be provided for different configurations: for example in OP-TEE there are different manifests for the different target architectures.

[1] `https://www.qemu.org/` (last visited 2018. 10. 22.)
[2] `https://github.com/aosp-mirror/tools_repo` (last visited 2018. 10. 22.)

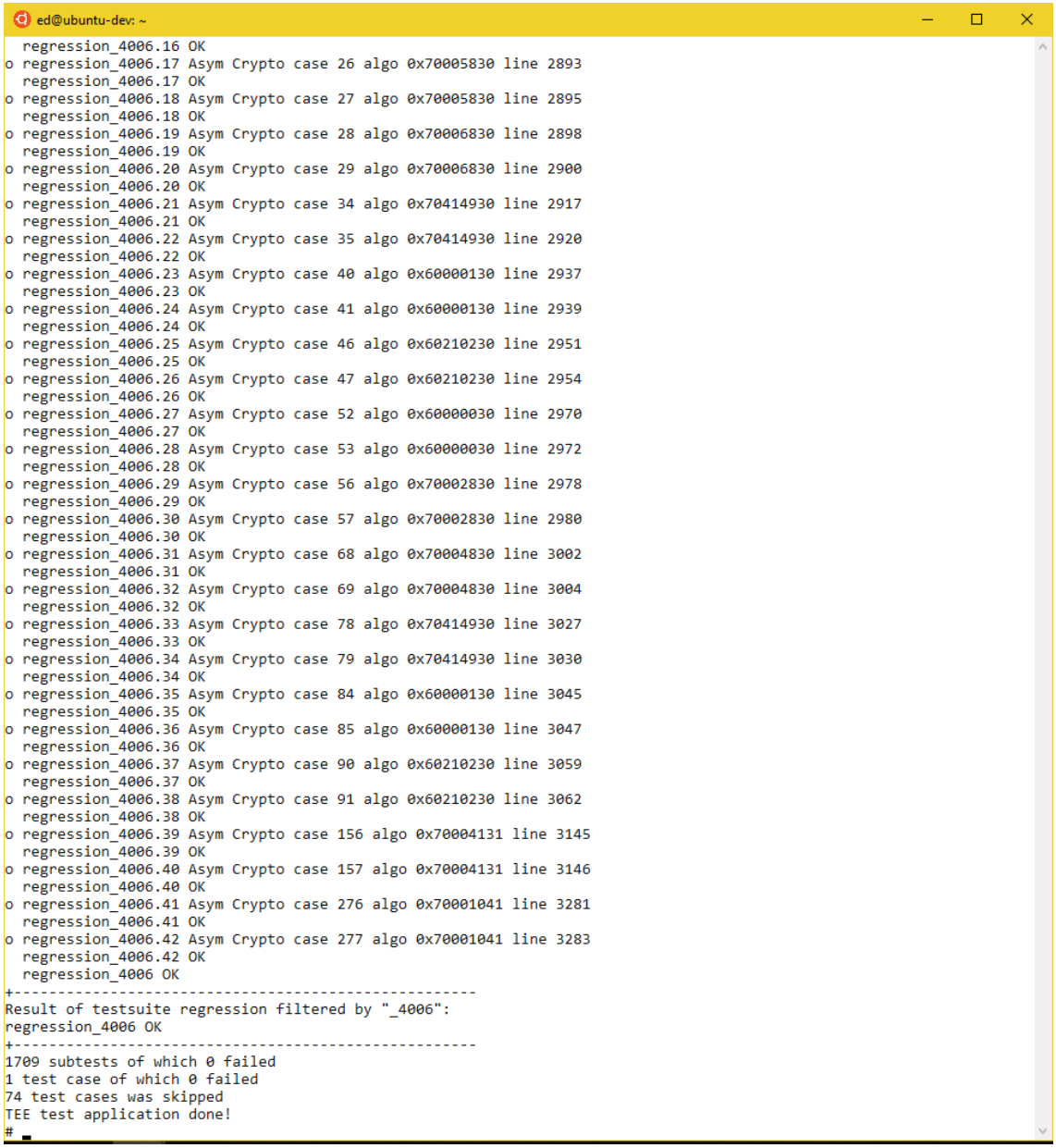


**Figure 3.1.** *Running OP-TEE regression tests with QEMU - NW*

It doesn't replace git, rather its goal is to make working with source code organized into numerous git repositories easier.

For OP-TEE, the manifests are stored on GitHub, in the `OP-TEE/manifest.git` repository. I used the `default.xml` to create a basis for developing SKS. Following the instructions in the `OP-TEE/build.git` repository, I could easily produce a working Linux plus OP-TEE system running in QEMU. Of course, this system didn't have the SKS and some other tools that I needed yet.

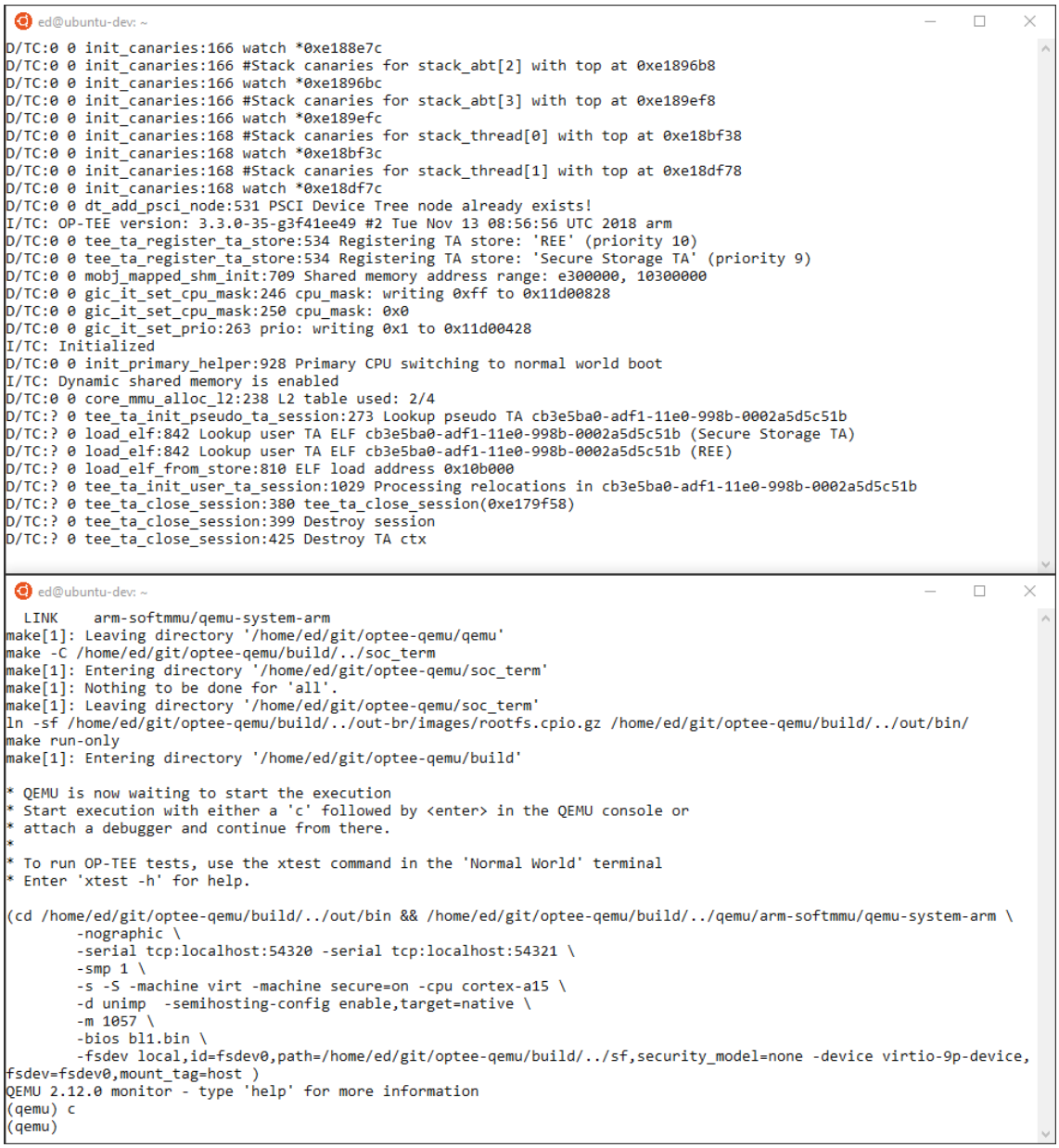


**Figure 3.2.** *Running OP-TEE regression tests with QEMU - SW and QEMU*

### Buildroot

Buildroot[3] is a tool that aims to make cross compilation and building custom Linux systems for embedded devices easier. It is basically a collection of makefile scripts and can be configured through various interfaces, e.g. menuconfig, just like the Linux kernel. As an output, buildroot produces a whole root filesystem, ready to be flashed onto the device and used. There is no package manager in the system created, so updating a single program is only possible by updating the whole system. There are many packages available in buildroot that can be included in the build, by simply selecting them in menuconfig, however it is also possible to create external packages, if something is needed that is not

[3] `https://buildroot.org/` (last visited 2018. 10. 22.)

already part of buildroot. This is done by creating two new files: first, the `Config.in` file describes how the package looks in menuconfig and what other packages it depends on and the `package_name.mk` file describes how to acquire the source and build it. There some other files that can be included (e.g.: `patchname.patch` for applying patches to the files contained by the package, `package_name.hash` for verifying the integrity of the package), but supplying the first two files is enough to create a working package.

Recently, OP-TEE has started to integrate its build process into buildroot, so adding the SKS and other needed tools had to be done through buildroot. I added the client and test part of sks, by simply replacing the source of optee_client and optee_test with the sks branch of the GitHub user etienne_lms. I added the SKS TA as an external buildroot package. OpenSSH was already part of buildroot, but the OpenSC pkcs11-tool had to be added as an external package as well.

## 3.2 SKS improvements

In this section, I will introduce my improvements on the SKS proposal, by going over the various problems I have encountered while trying to use it with OpenSSH.

### 3.2.1 Finding missing functions with pkcs11-tool and SSH

As mentioned in Section 2.2.1, the PKCS#11 standard is rather big, so often only parts of it are implemented. This is even more true for SKS, since it is still heavily under development. Because of this, I decided to enumerate the necessary additions, by trying to use ssh with SKS and checking where it would fail. Then implement or fix the specific functionality that caused the error and try again. This way I did not have to understand the whole codebase at once, or delve into the OpenSSH implementation either, rather I was able to gradually build up knowledge about these things as certain issues came up. As most of the API calls were already implemented in SKS, there were only a few cases where I had to implement a function from the ground up. In most of the cases, I only had to improve on already existing functions that were not working correctly in some situations. Of course, I collaborated with Etienne while working on this project, and contributed these additions and improvements to the project on GitHub. It might be worthy of mention here, that while I described the above process specifically with SKS in mind, my plans would have been the same, even if SKS did not exist. I would have patched a soft token, to print what PKCS#11 API calls are made while I use ssh and select a subset of the API to implement based on that. Doing this would have been likely harder to implement, so it is great that I had an already existing project to build on.

Specifically, I inserted the line in Listing 3.1 into the beginning of the handling function of each of the API calls, to easily get a sense of what is happening in the background when I give a terminal command.

```
1    printf("Function %s Entered\n", __func__);
```

**Listing 3.1.** *Print name of function that was called*

The parameter `__func__` is an implicitly declared identifier in the C language that expands to a cstring that has the name of the current function inside. A similar print can be placed at the end of the function to see when the function exits.

The following sections are the issues I encountered with this technique, roughly in chronological order. In each section, I detail the problem encountered and any relevant parts of the standard, then explain the solution I implemented for it.

### 3.2.2 Fixing the function list

One of the first problems manifested right at the beginning, when I started to explore the SKS soft token. The pkcs11-tool provides commands to get information about the devices that can be used with PKCS#11. The command that had problems was `--list-slots`. The output generated by the command can be seen in Listing 3.2, slightly edited for clarity.

```
# pkcs11-tool --module /lib/liboptee_cryptoki.so -L
    Function C_GetFunctionList Entered
    Function C_GetFunctionList End
    Function C_Initialize Entered
    Function C_Initialize End
    Function C_GetSlotList Entered
    Function C_GetSlotList End
    Function C_GetSlotList Entered
    Function C_GetSlotList End
Available slots:
Slot 0 (0x0):
    Function C_GetSlotInfo Entered
    Function C_GetSlotInfo End
OP-TEE SKS TA
    Function C_InitToken Entered
    Function C_InitToken End
C_GetTokenInfo() failed: rv = CKR_HOST_MEMORY
Slot 1 (0x1):
    Function C_GetSlotInfo Entered
    Function C_GetSlotInfo End
OP-TEE SKS TA
    Function C_InitToken Entered
    Function C_InitToken End
C_GetTokenInfo() failed: rv = CKR_HOST_MEMORY
Slot 2 (0x2):
    Function C_GetSlotInfo Entered
    Function C_GetSlotInfo End
OP-TEE SKS TA
    Function C_InitToken Entered
    Function C_InitToken End
C_GetTokenInfo() failed: rv = CKR_HOST_MEMORY
```

```
        Function C_Finalize Entered
        Function C_Finalize End
```

**Listing 3.2.** *PKCS#11 functions called when listing slots*

Based on the output, it seemed that for some reason, the pkcs11-tool thinks that it is calling the `C_GetTokenInfo`, but in reality the `C_InitToken` function is getting called. The reason for this error is rooted in the way the client applications acquire the function pointers to the PCKS#11 API, with the `C_GetFunctionList` method. This method returns a pointer to a `CK_FUNCTION_LIST` structure, which in turn contains function pointers to all the PKCS#11 API methods that the library has – even the unimplemented ones have to have stubs. According to the standard, this is inteded to be used in a way, so that applications can use shared PKCS#11 libraries easier and faster and to enable using more than one PKCS#11 library at once. In our case, some of the function pointers in the `CK_FUNCTION_LIST` structure were registered in the wrong order, which caused the pkcs11-tool to mix up the calls. The reason that this issue did not surface in the regression tests is probably that the same, out of order structure was used in the tests as well, meaning it could work together with the faulty SKS implementation, since they were wrong in the same way. Correcting the order of the functions in the structure resolved this behaviour.

### 3.2.3 Determining required buffer size

This problem came up while I was trying to gather information about the SKS soft token with the pkcs11-tool as well. When trying to list the available slots or mechanisms (`C_GetSlotList` and `C_GetMechanismList`), the program displayed the following error:

```
    error: PKCS11 function C_GetSlotList(NULL) failed:
        rv = CKR_BUFFER_TOO_SMALL (0x150)
    Aborting.
```

**Listing 3.3.** *Output of pkcs11-tool when listing slots*

After investigating a bit, I found that the pkcs11-tool was calling the respective API functions with a buffer pointer of `NULL` and zero length. According to the standard, this is done, in order to determine what size the buffer needs to be, in order for the result of the request to fit in it. The token should return with `CKR_OK` and set the buffer length for the necessary value. SKS already sets the length to the correct value, the only mistake was the wrong return value. I modified the logic, so that it returns with `CKR_OK` if the buffer pointer is `NULL` and this resolved the problem.

### 3.2.4 Key type inference in C_GenerateKeyPair

After exploring the soft token, the first step toward using for authentication with ssh is generating a new key to use on the token. I chose to use an RSA key, because I already had a little preliminary experience with RSA and it was also supported to some extent in

SKS. I used the pkcs11-tool with the following command to generate a 2048 bit RSA key with pkcs11-tool:

```
pkcs11-tool --module /usr/lib/libsks.so --label testtoken      \
    --login --pin 12341234 --keypairgen --label testkey        \
    --key-type rsa:2048
```

**Listing 3.4.** *Generate RSA key pair with pkcs11-tool*

However, the command resulted in pkcs11-tool displaying an error, stating that the `C_GenerateKeyPair` returned with `CKR_TEMPLATE_INCOMPLETE`. After some investigation, I found that the reason for this is that pkcs11-tool is not setting the `CK_KEY_TYPE` parameter for the call. This is because the standard states that it is not necessary to set this parameter, since it is implicit in the key generation mechanism, that is set with the `CK_MECHANISM` parameter. The reason for the error, is that this inference was not implemented in SKS.

Originally, I wanted to imlement the inference in the TA, however this was not possible, because the checks that determine whether an API call is well formed (i.e., in our case if it has every necessary parameter) are among the very first to run when the TA is called, and by the time I could insert the missing parameter, the call would already be rejected. Thus, I had to implement this in the shared library in Normal World.

In particular, what I decided to do, is scan the API call, before forwarding it to the TA, and if the `CK_KEY_TYPE` parameter is not specified in it, try to infer the missing information and insert it, but leave it as it is, if it is already present. Currently, the only inference is for the RSA key type, but other inferences can be very easily added, by inserting a new case in the switch case block that handles this logic.

### 3.2.5 C_FindObjects*

After generating the key pair we will use with ssh, we need to export the public key, so we can specify it for the server to accept it. At first I exported the RSA public key with the pkcs11-tool, but after applying all the changes listed here, it is also possible to use ssh-keygen to do this, which is also more convenient, because ssh-keygen takes care of the format conversions. Listing 3.5 shows the commands necessary to export with pkcs11-tool.

```
pkcs11-tool --module /usr/lib/libsks.so -r --type pubkey    \
            --label testkey > pub.key
openssl pkey -inform DER -pubin -in pub.key -out pub.pem
```

**Listing 3.5.** *Extract public key with pkcs11-tool*

And the necessary command to do this directly using ssh-keygen is in Listing 3.6.

```
ssh-keygen -D /usr/lib/libsks.so -e >> .ssh/authorized_keys
```

**Listing 3.6.** *Extract public key with ssh-keygen*

In order to extract the public key, we first need to acquire a handle for the public key object inside the token. This can be done with the `C_FindObjectsInit`, `C_FindObjects` and

`C_FindObjectsFinal` calls. The search parameters can be specified in `C_FindObjectsInit`, which also initializes the search. With each call to `C_FindObjects` the client can query one result of the search. To finish the active search, `C_FindObjectsFinal` has to be called.

In our case, the client application uses `C_FindObjectsInit` with no search parameters supplied, in which case all objects should be matched. This edge case was not correctly implemented in SKS, which meant that the client application could not find the public key. After adding code that correctly handled this case, the object handles were returned to the client.

### 3.2.6 Reading the public key with C_GetAttributeValue

Once they have the handle to the right object, the programs in the previous section need to extract the value of the public key object. This can be done with the `C_GetAttributeValue` function. This API call was not implemented before in SKS, so I had to build it from the ground up. When implementing this, there were two main aspects that had to be considered in order for this function to work correctly.

First, it is very important, that it is impossible to recover any values that are sensitive or unextractable. To accomplish this, I read the relevant parts of the standard, and also consulted the SoftHSM source code for a reference implementation, and created a function in the SKS TA, that decides whether a certain attribute can be disclosed.

The second is more related to our particular circumstances: when passing parameters in SKS from the Normal World to Secure World, they are serialized, and certain constants are translated from their respective numerical values in the PCKS#11 standard to the ones used internally by SKS. This serialization was implemented, however the deserialization and translation back to Normal World values was not, so it had to be added. I implemented these functionalities based on what their counterparts did, essentially reversing the serializing algorithm to get the deserializing algorithm.

In order to fully understand how the `C_GetAttributeValue` function works, a bit more information about PKCS#11 objects is necessary. In essence every object in PKCS#11 is a collection of attributes, for example `CKA_EXTRACTABLE` is a boolean attribute, and defines whether the given object is extractable. Another example is `CKA_MODULUS` that holds the modulus $N$ in an RSA public key object. To define it more precisely, the `C_GetAttributeValue` function obtains the value of one or more attributes of an object, in our case the modulus and public exponent attributes. The specification gives an algorithm to describe how to implement the `C_GetAttributeValue` function. Using the functionality described in the previous two paragraphs I implemented this algorithm, and successfully exported the public key.

### 3.2.7 Authenticating with OpenSSH

#### Helping OpenSSH find the private key

Having impelmented all of the above, we can set up an environment where we can actually start trying to use OpenSSH to establish a connection. We have a private key in our token, that is unextractable and we have inserted the public counterpart of that key into the `.ssh/.authorized_keys` file, so the server will accept it for authentication. For the sake of simplicity, I tested this on a single host, by connecting to localhost:

```
ssh root@localhost -I /usr/lib/libsks.so
```

**Listing 3.7.** *Connect as ssh client using SKS*

However, at first, when I gave this command, I was greeted with a password prompt. This meant, that the ssh server didn't accept any of the keys that the client offered, so it defaulted to authentication with a password. After some investigation, it became clear, that the OpenSSH client could not find the key pair on the SKS soft token. There are two reasons for this: the first is related to the login functionality in PKCS#11. It is possible to log in to a PKCS#11 token by supplying a PIN, and as one would expect, a logged in user can do some things that an unauthenticated user cannot. One of these differences is, that when searching for objects with the `C_FindObjects*` functions, the private key object is only among the results for an autenticated user. However, when I entered the ssh command, I was not prompted for a PIN number. OpenSSH decides whether it should prompt for a login PIN for the token, based on a flag in the token called `CKFT_LOGIN_REQUIRED`. If this flag is set, it means that the token has functionality that is only available to authenticated users. This flag was not set for the SKS token; after adding it, OpenSSH correctly prompted for the PIN and logged in to the token.

The second problem was caused by the method that is used by OpenSSH to identify what public keys and private keys on the token form key pairs. In PKCS#11, RSA key objects have an attribute called `CKA_ID`, that was not set by the `C_GenerateKeyPair` function in SKS, because the specification does not require it to be set. However, this is what the specification does say about `CKA_ID`:

> The CKA_ID attribute is intended as a means of distinguishing multiple public-key/private-key pairs held by the same subject (whether stored in the same token or not). (Since the keys are distinguished by subject name as well as identifier, it is possible that keys for different subjects may have the same CKA_ID value without introducing any ambiguity.)
>
> It is intended in the interests of interoperability that the subject name and key identifier for a certificate will be the same as those for the corresponding public and private keys (though it is not required that all be stored in the same token). However, Cryptoki does not enforce this association, or even the uniqueness of

> the key identifier for a given subject; in particular, an application may leave the key identifier empty. [15]

Reading this, I assumed that OpenSSH must be trying to use the `CKA_ID` to pair the public and private key object, but since the attribute is not set in our generated key pair, it discards these objects. To correct this, I amended the code in the SKS TA that handled generating key pairs with a section, that generates a random number and sets it as `CKA_ID` for both of the keys that are generated.

### The CKM_RSA_PKCS mechanism

The OpenSSH client creates a cryptographic signature with the private key of the user to authenticate to the server. In PKCS#11 this can be accomplished with the `C_SignInit`, `C_Sign`, `C_SignUpdate` and `C_SignFinal` API calls, depending on whether the the signing is done as single-part or as multi-part, with updates. The scheme that will be used is specified in the `C_SignInit`, with a `CK_MECHANISM_TYPE` type parameter. OpenSSH uses the `CKM_RSA_PKCS` mechanism, which is a multi-purpose mechanism based on the RSA PKCS#1 standard [10]. Among other things, it supports single-part encryption, decryption, signatures and verification. From here on, I will be writing about the case when it is used for signing. The exact scheme that this mechanism is based on is the RSASSA PKCS#1 v1.5, which in turn uses EMSA PKCS#1 v1.5 encoding. In the PKCS#1 specification, a rough outline of signing would be the following: hash the data that is to be signed, then use the paddings defined in the standard to generate the encoded message, where the interesting part for us is that the resulted encoded message will contain (with padding) the identifier of the hash algorithm used and the hash itself. Then apply the necessary cryptographic primitives to the encoded message to get the signature itself. However, the PKCS#11 specification states, that the `CKM_RSA_PKCS` mechanism only corresponds to the part of PKCS#1 v1.5 that involves RSA and does not compute a message digest. In accordance with this, it takes a message digest (generated by the client application) as input, and it does not and can not include the identifier of the hash algorithm that was used. In SKS, signing and verification with the `CKM_RSA_PKCS` mechanism was not implemented, because there is no corresponding function in the GPD TEE Cryptographic Operations API. There are algorithm identifiers for signing a message digest with PKCS#1 RSASSA, but those also require a hash algorithm, since the format of these is `TEE_ALG_RSASSA_PKCS1_V1_5_<hash_algorithm>`, so for example `TEE_ALG_RSASSA_PKCS1_V1_5_SHA1` would expect a message digest created by `SHA1` and include its algorithm identifier in the signature.

### OP-TEE extension TEE_ALG_RSASSA_PKCS1_V1_5

Fortunately, since OP-TEE is open source, it is possible to modify the source code and extend the API defined in the GPD specification. Behind the internal core API in OP-

TEE, the cryptographic operations are implemented using LibTomCrypt[4], an open source cryptographic library written in the C language. So the solution is self evident from here: define a new algorithm identifier in the API and connect it to the right LibTomCrypt function. The only problem was, that after looking at the source of LibTomCrypt in OP-TEE, it seemed that there was no option to do this in LibTomCrypt either, since even the LibTomCrypt API required a hash algorithm to be specified. At the same time, I noticed, that the timestamps in the LibTomCrypt source that was included in OP-TEE were somewhat outdated, so I inspected the GitHub repository of LibTomCryptI where I found that the newer version does support the same mode of operation that I need. To be exact, they added a new possible value for the padding parameter in the `rsa_sign_hash_ex` function, called `LTC_PKCS_1_V1_5_NA1` that allows this. At first, I wanted to update the whole LibTomCrypt library in OP-TEE to the latest version, but after comparing the differences between the two versions, I realized that OP-TEE had many minor modifications (e.g. for optimization purposes) that would have made the update process very lengthy and prone to errors, so I opted to only update `rsa_sign_hash_ex` and `rsa_verify_hash_ex` functions, while being very careful to keep the minor changes added by the OP-TEE developers in them. With the updated version of LibTomCrypt, I was able to add the new algorithm identifier to the Cryptographic Operations API and in turn connect it with the `CKM_RSA_PKCS` mechanism. And with that, OpenSSH was finally able to authenticate and connect to the server.

As it was a requirement for merging my work into the OP-TEE project, I created a regression test case for the newly added TEE API method. I created the new test cases by modifying test vectors already used by the OP-TEE test suite (originally from the US NIST Computer Security Resource Center[5]). Specifically, I kept the plaintext and key pair of the test vector and generated the signature with a python script. Then I inserted these values into their place and added them to the list of test vectors. The python code that I used to execute these steps is in Appendix A.1.

As a side note here, the question has arisen in me, whether it makes any sense to even have such a mechanism as `CKM_RSA_PKCS`, since if the hash algorithm is not included in the signature, how would the receiver of the message know how to verify the the message with the signature, without knowing how to create the right message digest from it. A simple possibility that gets around this problem, is if both parties in a communication follow a protocol that specifies what hash function they have to use, so the hash function—although not explicitly specified in the signature—is known to the receiver. I have also been able to find some other examples of cases where this functionality is required. The first one is mentioned in the commit message of the change[6] in which the `LTC_PKCS_1_V1_5_NA1` option was added to LibTomCrypt and according to the author, the early versions of the SSL protocol did not set the hash algorithm identifier in `SERVER_EXCHANGE_MESSAGE`

[4] `https://www.libtom.net/LibTomCrypt/` (last visited 2018. 10. 22.)

[5] `https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/digital-signatures#rsavs` (last visited 2018. 10. 22.)

[6] `https://github.com/libtom/libtomcrypt/commit/aa4bae5ae9a2` (last visited 2018. 10. 22.)

messages, hence the feature was added to LibTomCrypt to support the format. The same feature was requested[7] by someone else who was also implementing a PKCS#11 soft token. The last example is in the `pyca/Cryptography` python module, where it was requested in an issue[8] on GitHub by a contributor for the TOR project, because it was necessary for compatibility reasons, although as far as I can tell, in this case the functionality was not added.

[7] `https://lists.randombit.net/pipermail/botan-devel/2008-November/000696.html` (last visited 2018. 10. 22.)

[8] `https://github.com/pyca/cryptography/issues/3713` (last visited 2018. 10. 22.)

# Chapter 4

# Evaluation

## 4.1 Basic functioning

In my paper, I have extended the SKS OP-TEE proposal, making it possible to use it with an OpenSSH client to authenticate and log in to an ssh server. This can be achieved for example by using the commands mentioned through Section 3.2 and is demonstrated in Listing 4.1.

```
# pkcs11-tool --module /usr/lib/libsks.so --init-token  \
              --label testtoken --so-pin 12341234
Using slot 0 with a present token (0x0)
Token successfully initialized

# pkcs11-tool --module /usr/lib/libsks.so --label testtoken  \
              --login --so-pin 12341234 --init-pin --pin 12341234
Using slot 0 with a present token (0x0)
User PIN successfully initialized

# pkcs11-tool --module /usr/lib/libsks.so --label testtoken  \
              --login --pin 12341234 --keypairgen            \
              --label testkey --key-type rsa:2048
Using slot 0 with a present token (0x0)
Key pair generated:
Private Key Object; RSA
    label: testkey
    ID: 01bebf99
    Usage: decrypt, sign, unwrap
Public Key Object; RSA 2048 bits
    label: testkey
    ID: 01bebf99
    Usage: encrypt, verify, wrap

# ssh-keygen -D /usr/lib/libsks.so -e > .ssh/authorized_keys
# chmod 0600 .ssh/authorized_keys

# ssh root@localhost -I /usr/lib/libsks.so
```

```
The authenticity of host 'localhost (127.0.0.1)' can't be established.
ECDSA key fingerprint is SHA256:KETZVgw2Otax58AKVxy2pWHB/w8mvsywB2EG9nO3iGM
    .
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (ECDSA) to the list of known hosts.
Enter PIN for 'testtoken':
# ls /
bin etc lib32 mnt root sys var
data init linuxrc opt run tmp
dev lib media proc sbin usr
# exit
Connection to localhost closed.
#
```

**Listing 4.1.** *Using SKS with OpenSSH*

Alternatively, for the public key extraction pkcs11-tool could also be used. I have also tested this using the ssh-agent to handle the token:

```
# ssh-add -s /usr/lib/libsks.so
Enter passphrase for PKCS#11:
Card added: /usr/lib/libsks.so

# ssh root@localhost
# exit
Connection to localhost closed.
```

**Listing 4.2.** *Using SKS with ssh-agent*

## 4.2 Performance

As usual, adding security features creates an overhead and hinders performance. In the case of SKS, the overhead is the context change between Normal World and Secure World and the serialization and deserialization required to pass the parameters. However, because of the nature of the ssh protocol, this overhead is limited to the key generation and the authentication process [17], since after authentication temporal keys are used for communication, so it will not affect overall system performance that much. Testing on QEMU, the time required for authentication was not noticably different: connecting without SKS took around 1.6 seconds while it took 3 seconds with using SKS, but it's hard to take an exact measurement, since the program requires user input. On the other hand, the key generation took significantly longer with SKS: it took around 5 seconds to generate a 2048 bit RSA key with `ssh-keygen`, but it took as long as 53 seconds to do the same through the pkcs11-tool with SKS. This could possibly be because SKS is generating its own randomness, but I have not done any investigation to find out what takes so long during key generation. The measurements were taken with the `time` linux utility.

## 4.3 Security

The security of SKS depends on TrustZone and OP-TEE working correctly, and having correct configurations for the device that is being used. If all these are right, the secret key can never be extracted from the SKS soft token. So even if an adversary takes complete control over Linux, they can only use it for certain cryptographic operations, but can not recover it. Of course, the PKCS#11 standard is fairly big, and as mentioned in [4], there have been vulnerabilities found in it before. So, to further limit the attack surface, one could evaluate what PKCS#11 functions the client applications used in their system need, and disable the rest in the SKS TA.

## 4.4 Limitations

SKS is still heavily under development, and not yet merged into the mainline OP-TEE repository, so it has to be added by hand if we want to use it. Another limitation is, that I only tested it with OpenSSH and pkcs11-tool, so if someone would want to use a different application that has PKCS#11 support, they would likely discover some issues, just like I did. Another limitation, that is a result of using OpenSSH. Since in my implementation, only the private key of the client is protected by SKS, but the fingerprint of the ssh server is stored in Normal World, if an adversary were to take over the REE side of the system, it could modify it and impersonate the ssh server.

# Chapter 5

# Conclusion

In this paper, my aim was to protect cryptographic secrets on an IoT device considering the possibility that it may even be physically compromised, as a result of its location likely being unsecurable. Low price was also an objective, in order to raise the likeliness of adoption, so security co-processors were ruled out. Building on TrustZone technology, OP-TEE and the SKS proposal I improved the SKS proposal in multiple ways to make it possible to use SKS to store OpenSSH RSA identities and use them to authenticate to a remote server without Linux or OpenSSH actually accessing the keys.

The OP-TEE API extension in Section 3.2.7 has been merged into the OP-TEE project on GitHub as a pull request[1], and the various other chnages in SKS have also been contributed through pull requests.

As already mentioned, SKS still needs much more development. As a continuation of the approach in this paper, it would be interesting, to ensure compatibility with a VPN solution, since that is a functionality that an IoT device could also use to secure its communications.

[1] `https://github.com/OP-TEE/optee_os/pull/2524` (last visited 2018. 10. 22.)

# Acknowledgment

I would first like to thank my thesis advisor Dr. Levente Buttyán of the CrySyS laboratory at the Budapest University of Technology and Economics. Dr. Buttyán provided me with valuable advice, guidance and encouragement. I would also like to thank Etienne Carrière for allowing me to use the SKS proposal and for reviewing my modifications to it.

# List of Figures

# Listings

# Bibliography


[1] Arm security technology – building a secure system using trustzone technology. Technical Report PRD29-GENC-009492C, 2009.

[2] Manos Antonakakis, Tim April, Michael Bailey, Matthew Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, pages 1093–1110, Berkeley, CA, USA, 2017. USENIX Association.

[3] Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. Attacking and fixing pkcs#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 260–269, New York, NY, USA, 2010. ACM.

[4] Christian Cachin and Nishanth Chandran. A secure cryptographic token interface. In *Proceedings of the 2009 22Nd IEEE Computer Security Foundations Symposium*, CSF '09, pages 141–153, Washington, DC, USA, 2009. IEEE Computer Society.

[5] Jolyon Clulow. On the security of pkcs# 11. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 411–425. Springer, 2003.

[6] Tim Cooijmans, Joeri de Ruiter, and Erik Poll. Analysis of secure key storage solutions on android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '14, pages 11–20, New York, NY, USA, 2014. ACM.

[7] Szilárd Dömötör, Márton Juhász, Gábor Székely, and István Telek. Enhancing the security of the internet of things: Design and implementation of security mechanisms for embedded platforms. In *BME Students' Scientific Conference (TDK)*, 2018.

[8] GlobalPlatform. *TEE Internal Core API Specification*, 2016. Version v1.1.2.

[9] GlobalPlatform. *TEE System Architecture*, 2018. Version v1.2.

[10] J. Jonsson and B. Kaliski. Public-key cryptography standards (pkcs) #1: Rsa cryptography specifications version 2.1, 2003.

[11] Georgios Kambourakis, Constantinos Kolias, and Angelos Stavrou. The mirai botnet and the iot zombie armies. In *Military Communications Conference (MILCOM), MILCOM 2017-2017 IEEE*, pages 267–272. IEEE, 2017.

[12] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014.

[13] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[14] OASIS. *PKCS #11 Cryptographic Token Interface Usage Guide*, 2014. Version 2.40.

[15] OASIS. *PKCS #11 Cryptographic Token Interface Base Specification*, 2016. Version 2.40 Plus Errata 01.

[16] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted Execution Environment: What It is, and What It is Not. In *14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, Helsinki, Finland, August 2015.

[17] T. Ylonen and Ed. C. Lonvick. The secure shell (ssh) authentication protocol, 2006.

# Appendices

## A.1 Python code used for test case generation

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives.serialization import load_pem_private_
    key
from cryptography.hazmat.backends import default_backend
from cryptography.utils import int_to_bytes, int_from_bytes
import binascii

pk = open('key.pem', 'rb')
key = load_pem_private_key(pk.read(), password=None, backend=default_
    backend())
message = bytes.fromhex( 'e567a39ae4e5ef9b6801ea0561b72a5d4b5f385f0532fc9fe
    10a7570f869ae05c0bdedd6e0e22d4542e9ce826a188cac0731ae39c8f87f9771ef
    02132e64e2fb27ada8ff54b330dd93ad5e3ef82e0dda646248e35994bda10cf46e5abc
    98aa7443c03cddeb5ee2ab82d60100b10296318979702 75f119d05daa2220a4a0defba
    ')
# ========================
signature = key.sign(
    message,
    padding.PKCS1v15(),
    hashes.SHA256()
)
print( binascii.hexlify(signature) )
# ======================== recover ct
modulus = key.public_key().public_numbers().n
public_exponent = key.public_key().public_numbers().e

sig_as_bytes = signature
sig_as_long = int_from_bytes(sig_as_bytes, byteorder='big')  # convert
    signature to an int
blocksize = len(sig_as_bytes)

decrypted_int = pow(sig_as_long, public_exponent, modulus)

# convert the int to a byte array
decrypted_bytes = int_to_bytes(decrypted_int, blocksize)
decrypted_hex = binascii.hexlify(decrypted_bytes)
print( decrypted_hex )
```

```
# ======================== remove hash oid
sha256 = bytes.fromhex('3031300d060960864801650304020105000420')
new_msg = decrypted_bytes.replace(b'\x00'+sha256, len(sha256)*b'\xff'+b'\x
    00')
print( binascii.hexlify(new_msg) )

# ======================== reencrypt
private_exp = key.private_numbers().d
new_msg_int = int_from_bytes(new_msg, byteorder='big')
new_msg_encrypted_int = pow(new_msg_int, private_exp, modulus)
new_msg_encrypted_bytes = int_to_bytes(new_msg_encrypted_int, blocksize)
print( binascii.hexlify(new_msg_encrypted_bytes) )

# ======================== test to recover ct
sig_as_bytes = new_msg_encrypted_bytes
sig_as_long = int_from_bytes(sig_as_bytes, byteorder='big')
blocksize = len(sig_as_bytes)

decrypted_int = pow(sig_as_long, public_exponent, modulus)

# convert the int to a byte array
decrypted_bytes = int_to_bytes(decrypted_int, blocksize)
decrypted_hex = binascii.hexlify(decrypted_bytes)
print( decrypted_hex )
```