Master's Thesis

A Development Environment for ARM TrustZone with GlobalPlatform Support

Fredrik Kvant
Max Kellner

Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, June 2014.

A Development Environment for ARM TrustZone with GlobalPlatform Support

Fredrik Kvant and Max Kellner

Department of Electrical and Information Technology
Lund University

Advisor: Martin Hell

June 15, 2014

Printed in Sweden
E-huset, Lund, 2014

Abstract

TrustZone is a way to provide security in devices built on the ARM platform by separating sensitive software from user installed content. The purpose of this thesis is to present an easier and cheaper way of testing applications making use of the TrustZone technology. Using a collection of open source software an emulator capable of testing trusted applications and their clients is developed. By implementing the Global Platform TEE client and TEE internal API specifications we hope that this work will be useful to anyone developing applications implementing these open specifications. To facilitate simple debugging of applications, extra efforts have been spent on providing useful error logs. Instructions and scripts for how to run and utilize GDB with the emulator are also included.

Acknowledgements

First we would like to thank our advisor Martin Hell at LTH for his assistance with the administrative work during this thesis. We would also like to thank everyone at Sony Mobile who gave us the opportunity to perform our thesis. In particular we give our thanks to Stefan Andersson and Petter Wallin for taking time to have weekly meetings to discuss the progress of our work. Their surprise of our accomplishments gave us much confidence. We are also thankful to Pravat Dalbehera for helping us with everything administrative at Sony. Finally we would like to thank Daniel Sangorrin Lopez for taking time to answer questions regarding the SafeG monitor and FMP during the early stages of our work.

Table of Contents

Chapter 1
Introduction

1.1 Motivation

Along with an ever increasing market of smart devices comes security issues. Devices that before never or rarely had to deal with security issues are now target of attacks. Devices such as smartphones and tablets are used for both sensitive secure actions as well as entertainment. Company devices may contain sensitive data. A device may contain DRM protected media which malicious parties may try to access without permission. Other malicious parties could attempt to steal sensitive information or bank credentials. Some everyday users may gain root access to their device which might compromise security of the device.

One way to combat these security problems is hardware isolation mechanisms. ARM TrustZone is a hardware isolation mechanism that can be used to improve security in systems using the ARM processor architecture, an architecture that is widely used in mobile devices. There is currently a lack of open hard- and software available to developers of secure applications making use of ARM TrustZone. In order to encourage development of more secure applications an open emulator with which developers could test both rich and trusted applications developed with TrustZone in mind would be of use.

1.2 Goal

The main goal of this thesis is to provide an open emulator for testing and development of client and trusted applications working in conjunction on platforms using TrustZone. By adhering to the Global Platform specifications we seek to make this work as widely usable as possible. Listed below are important points that must be taken into consideration.

- Set up an emulator with TrustZone
  - Does any open emulator currently provide any form of TrustZone support?
  - How can communication between rich and trusted applications be done?
- Integrate a subset of the GlobalPlatform API within the emulator

  - Implement a subset of the Global Platform specification
  - Test applications using the specification
- Add the ability to debug rich and trusted applications in the emulator
  - If possible, the use of a debugger, such as GDB, should be easy
  - Otherwise provide an alternative way to debug applications
- Be more than a proof of concept
  - Provide a manual for the usage of the emulator
  - Provide example code that shows functionality

1.3 Terminology

TEE, Trusted Execution Environment: The Trusted Execution Environment is the secure area within a device. It consists of both hardware and software. Sensitive applications and data is stored in the TEE. Trusted Execution Environment, Secure World and Trusted World are used interchangeably.
REE, Rich Execution Environment: The Rich Execution Environment is the normal area with a device. The rich OS and user installed applications are run in the REE. Rich Execution Environment, Normal World and Non Trusted World are used interchangeably.
Normal World: See REE
Non Trusted World: See REE
Secure World: See TEE
Trusted World: See TEE
TA, Trusted Application: A trusted application is an application which runs in the TEE and receives operations from security aware application in the normal world.
Trusted OS: The trusted OS is a part of the TEE and is usually very small, containing the bare essentials.
Rich OS: The rich OS is a part of the REE and is the operating system the user is interacting with.
Monitor: The monitor is a small piece of software in the secure world that manages the switch between worlds.
Root of Trust: A Root of Trust is an entity defined as trusted. It can be used to decide if other entities should be trusted or not.
Secure Boot: A Secure Boot is achieved when only software and firmware trusted by the Root of Trust is executed during boot.

1.4 Structure

The remainder of this report is structured as follows: In chapter 2, we will begin by defining isolation mechanics, we will introduce ARM and specifically TrustZone and we will describe works used in this thesis. In chapter 3, we will describe how our work was structured, how all previous works were integrated and what

sections of GlobalPlatform were implemented. In chapter 4 we will present what functionality has been added by this work and what usages it provides as well as take a look at the limitations our implementation has. Finally, in chapter 5 we will give a short overview of the entire report and mention a few examples of what can be done to improve our work.

Chapter 2
Background

2.1 Isolating a Trusted Execution Environment

2.1.1 Introduction

The security of a device can be increased by preventing user installed applications from accessing sensitive data or memory locations within the device. If malicious software within a device is not able to access memory it is not able to change the behaviour of other applications. If malicious software is not able to access keys, they cannot be reported to a third party. By separating a system into two (or more) distinct sections where one is specifically used for secure system functions it is possible to achieve strong isolation. This is done by allocating resources critical to security functions to a separate trusted operating system which is in control of security.

A Trusted Execution Environment, or TEE, is made up of both software and hardware working in conjunction to protect against software attacks from within the rich OS. The software within the TEE is run in parallel with the rich OS and provides security services to the rich OS. The TEE can for example be used to protect sensitive applications by isolating them from the rich OS and it can protect drivers or information needed for the functionality of the device. The security a TEE provides is dependent on how it is realized; a TEE implemented with TrustZone is susceptible to hardware tampering while one done with a Secure Element is not.

2.1.2 Secure Element

A secure element, or SE for short, is software working in conjunction with a piece of hardware specifically designed to be tamper resistant [1]. The role of the SE could be to store, generate and protect cryptographic keys, perform crypto operations, store protected data or to provide a root of trust in order to protect the integrity of the system.The secure elements can come in one of many variations such as a Universal Integrated Circuit Card (UICC), an embedded SE or a secure microSD. The Trusted Platform Module [2], or TPM for short, is a type of embedded SE. The main drawback of using an SE is that additional hardware is needed, increasing the cost of production.

2.1.3 Hypervisors

A hypervisor is a piece of software, firmware or hardware capable of running several operating systems on the same hardware system. The hypervisor, also known as a virtual machine monitor, creates and manages virtual machines. The system that runs the hypervisor is defined as the host machine and each virtual machine as a guest machine. The hypervisor runs with a higher privilege level than the guest systems and distributes resources of the host system between guest machines. This way hardware is shared between guest machines, but they remain isolated from each other. Hypervisors are mostly used in cloud computing to create and run a multitude of remote systems, but the isolation they provide make them useful for security reasons.

The drawback of hypervisors is that they generate some overhead for each system call as each call needs to be handled in the hypervisor and require some modification to the OS running within the guest machine [9]. Additional overhead is especially problematic within an embedded and/or mobile system where resources might be scarce and battery-life has to be taken into account.

Examples of hypervisors include the VMware vSphere Hypervisor[4] and the Xen Project Hypervisor[3]

2.1.4 Trustzone

TrustZone is a technique that can virtually separate hardware into two domains referred to as "worlds". In this manner it is similar to a hypervisor, but it differs in the way that both worlds potentially have full access to all hardware while it is still possible to restrict access to certain resources for one of the worlds. Each world runs its own operating system.

The advantage of TrustZone is that no additional security hardware, such as a cryptoprocessor, is needed. The trusted OS can execute with the same processing power as the normal OS without needing another processor. By exposing an API to applications in the rich OS it is possible for them to make use of the security features of the TEE.

Most mobile devices today contain an ARM processor[5] and all of the Cortex-A Series processors support TrustZone. Smartphones and netbooks typically utilize this line of processors. Below are a few usage examples for TrustZone as listed by ARM [6] :

- Secured PIN entry for enhanced user authentication in mobile payments and banking
- Protection against trojans, phishing and Advanced Persistent Threats
- Enable deployment and consumption of high-value media (DRM)
- BYOD (Bring your own device) device persons and application separation
- Software license management
- Loyalty-based applications
- Access control of cloud-based documents

- e-Ticketing Mobile TV

Security of a TrustZone system is completely dependent on the security of the secure monitor and a secure boot; if any of these are compromised security can no longer be guaranteed. Security also depends on the trusted applications that are run in the secure world. If an application containing an exploit or bug gets signed and installed in the TEE it could compromise keys and sensitive data that it can access. Depending on the implementation of the TEE, a rogue trusted application could jeopardize everything stored in the secure world.

2.2 The ARM Processor

2.2.1 Introduction

The ARM processor is a RISC-based processor. RISC (Reduced Instruction Set Computing) opposed to CISC (Complex Instruction Set Computing) is a simple instruction set with the goal of reaching higher performance by utilizing simpler instructions that can execute at a higher rate. Such processors use significantly fewer transistors than traditional processors, reducing manufacturing costs, heat generation and power consumption. The RISC-architecture is the most widely used in embedded systems. In this chapter the ARM processor will be quickly introduced.

2.2.2 Interrupts

ARM processors are able to handle two different kinds of interrupts, the normal interrupt request, IRQ, and a fast interrupt request, FIQ. They differ in that the fast interrupt request has a higher priority than the normal interrupt request. Interrupts are disabled while a FIQ is being handled, thus a FIQ is able to interrupt the handling of an IRQ while the opposite is not true.

In TrustZone, devices designated as secure make use of FIQ while normal devices make use of IRQ. This means that the secure world is able to take control from the normal world, but the normal world is not able to take control away from the secure world.

2.2.3 Processor Modes

The ARM processor can run in several different modes. As of version 7 the following modes are available:
User mode: User mode: The non-privileged mode. This is the mode used during normal usage.
Fast Interrupt mode: Fast Interrupt mode: This mode is entered whenever the processor accepts an FIQ interrupt.
Interrupt mode: Interrupt mode: This mode is entered whenever the processor accepts an IRQ interrupt.
Supervisor mode: Supervisor mode: A privileged mode entered whenever the CPU is reset or when a SVC(Supervisor Call) instruction is executed.

Abort mode: Abort mode: A privileged mode that is entered whenever a prefetch abort or data abort exception occurs.
Undefined mode: Undefined mode: A privileged mode that is entered whenever an undefined instruction exception occurs.
System mode (ARMv4 and above): System mode (ARMv4 and above): The only privileged mode that is not entered by an exception. It can only be entered by executing an instruction that explicitly writes to the mode bits of the CPSR.
MON mode (TrustZone only): MON mode (TrustZone only): A mode that executes the secure monitor in TrustZone systems. The monitor will be introduced in the next chapter. This mode is only available to the secure cpu and is entered when switching from one world to the other.

2.2.4 Registers

The ARM processors hold 16 32-bit registers. Some are given aliases:

- R13 is also referred to as SP, the Stack Pointer.
- R14 is also referred to as LR, the Link Register.
- R15 is also referred to as PC, the Program Counter.

Registers are banked depending on CPU mode. Banking is the act of switching one set of registers for another. Registers 0-7 are never banked, 8-12 are banked for FIQ, 13-14 are banked for every mode. Register 15, the program counter, is never banked. This means that every mode has its own stack pointer and link register and that while in FIQ mode, the cpu has access to a separate set of registers r8 to r12. In addition to these 16 registers there is a CPSR (Current Program Status Register) for the current mode as well as a SPSR (Saved Program Status Register) banked for every cpu mode.

2.2.5 Coprocessors

The ARM architecture supports up to 16 coprocessors, which can be used to extend the instruction set. Coprocessor 15, referred to as the system control coprocessor, is reserved for control functions such as managing the cache, MMU and system settings. This coprocessor is a set of registers that can be written to and read from [7]. One register in particular is important to know about, the Secure Configuration Register, which contains the "Not Secure" bit which will be introduced in the next section.

2.3 Important Aspects of the TrustZone Technology

2.3.1 Secure World

TrustZone separates the device into two domains referred to as worlds; the Normal World and the Secure World. The Secure World acts as a TEE, see section 2.1.1, for the system. It is a separate execution environment which has its own OS, drivers and hardware.

The software run in the Secure World may be bare-metal, containing nothing but security libraries, or it can be a full-fledged OS. Since most platforms provide limited resources, mostly in in terms of memory, to the Secure World, it tends to be small in size and only contain the bare essentials.

The OS in the secure world can be viewed as if it is run in parallel with the Rich OS. On a multi-core processor this may often be the case as one core could be dedicated to run in secure mode. Execution on the other cores continues even if there has been a switch to the secure world.

The OS in the secure world loads and runs trusted applications, or TAs. Trusted applications are central to the usefulness of TrustZone, they are the endpoint where your calls from a client application within the rich OS ultimately reaches. In short: the secure world provides a safe execution environment and the trusted applications contain the functions you want to execute securely. Trusted applications can come pre-installed from the factory or they can be installed during the lifetime of the device if the implementation allows for it. Generally trusted applications must be signed by the manufacturer in order to be installed.

As mentioned in the start of this chapter, the Secure World has its own hardware. This does not mean that there are two of every component but rather that some parts of the hardware are only accessible from the Secure World. An example would be the section of the memory used by the secure world. This section of memory is often referred to as secure memory and inaccessible elsewhere; the Rich OS doesn't even know it exists. Another example of this is storage; most devices with TrustZone provide some sort of secure storage that only the Secure World can know about. If you pre-load the secure storage with a number of private keys it is possible to let trusted applications perform signing without the need to expose keys to the user. Different forms of using securely stored private keys to sign or verify data is one of the most common usages of TrustZone today.

2.3.2 Normal World

Calling one world secure implies that the other is unsecure. While sometimes referred to as Non-Trusted, the normal world is not necessarily insecure, it is simply less secure, and more focused on presenting a good user experience, than the dedicated secure world. It is less secure in that the user is able to install content which might compromise the security of the device, whereas the inclusion of content into the secure world is tightly regulated.

The normal world runs the traditional OS, the Rich OS, which the user interacts with. Common examples include Windows, Linux and Android. Applications that wish to utilize secure services can call on functions made public via a TrustZone API. Applications can request that data be encrypted/decrypted by the secure world which in turn, encrypts the data and returns the encrypted/decrypted data to the application in the normal world. This way, the keys used are never exposed to the normal world.

The normal world can be seen as an isolated environment, with a lower privilege level, within the secure world. Figure 2.1 illustrates how the normal world is an environment within the secure world. When TrustZone is not enabled, the only "world" that remains is the secure world, not the normal world. Because there is

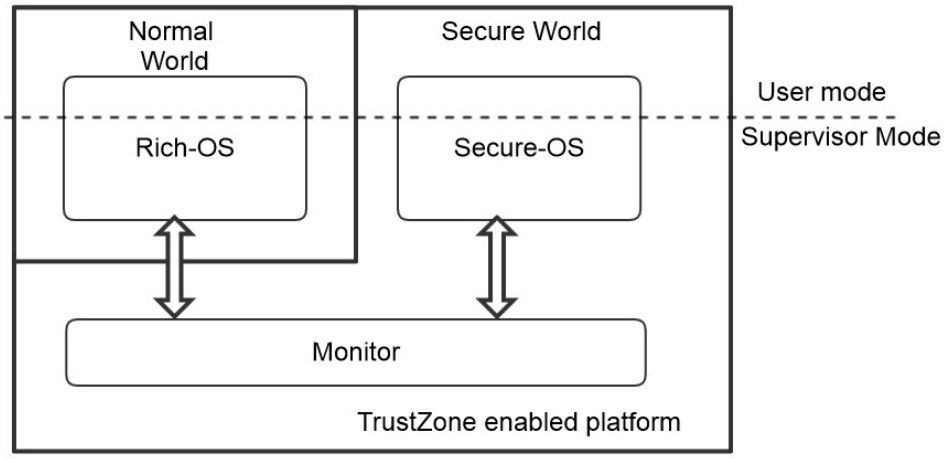

Figure 2.1: The relationship between the secure world and the normal world in a TrustZone system

no isolated section within the system however, the additional security protection disappears.

2.3.3 The NS-bit

The NS-bit, short for not secure, located in the secure configuration register, informs the system which mode the processor is currently running in. The value of this NS-bit is propagated throughout the system via the AMBA3 AXI bus[8] and hardware control determines if the system is allowed to access certain resources. If the NS-bit is set to 1, the processor is working in a non secure mode and is not able to access secure memory or devices. If the NS-bit is set to 0, the processor is working in a secure mode and is able to access any hardware within the system.

2.3.4 Virtual Processors

When running TrustZone the processor is divided into two virtual processors, one trusted and one untrusted. If the processor has several cores one could be dedicated to run as trusted while the remaining cores run as untrusted. Each virtual processor have their own memory management unit and, unless one or more cores have been dedicated to the secure world, the full capability of the physical processor. Both processors can also run in the first seven modes described in section 2.2.3. The eighth mode, monitor mode, which runs the secure monitor, is only available to the trusted virtual processor.

2.3.5 The Monitor

The most important aspect of TrustZone is the monitor. The secure monitor is a small piece of software which will make sure neither processor will have access to

registers belonging to the other after a switch between the worlds. The monitor also prevents the untrusted processor from using secure memory or devices by managing the NS-bit. It is important to note that the secure monitor is only executed while in monitor mode and only the trusted processor can run in monitor mode. The monitor can be entered from software via the Secure Monitor Call, SMC, when in supervisor mode or via hardware interrupts.

Switching to the secure world can be compared to a regular context switch. The monitor stores all registers of the currently running world and loads the previously stored registers of the world it is switching to. Because the monitor is trusted and run in the secure world, it can store the secure world's register information in its own address space without risk of leaking this information.

For client applications to access functions of trusted applications the secure monitor call must be used in order to enter the monitor and switch to the secure world. The secure monitor call can only be used when in supervisor mode, thus a supervisor call, SVC, must first be made. When returning to the normal world, the same procedure must be followed. Sometimes, if the secure world does not have a user mode, a supervisor call may not be necessary. Figure 2.2 displays how sending data to a more privileged mode requires an SVC or SMC to be made. Arrows with no command noted shows how data flows to sections with lower privilege levels.

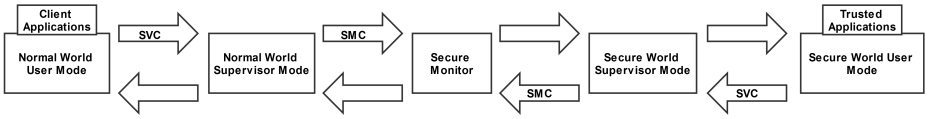

Figure 2.2: The flow of information within a system using TrustZone.

2.3.6 Hardware Partitioning

With TrustZone the physical memory is partitioned into two distinct regions; trusted and untrusted memory. Untrusted memory can also be registered as shared memory, which can be used by both worlds to communicate with each other.

The normal world has access to untrusted memory which it may register and use as shared memory to communicate with the secure world. The secure world on the other hand has full access to memory. In general however, it uses only secure and shared memory. What memory belongs to which region can be controlled via the TrustZone Protection Controller [11] which is only accessible from the secure world.

Like memory, devices connected to the TrustZone-aware system are divided into secure and normal devices. The normal world is blocked from using devices designated as secure. In general secure devices generate FIQs and non-secure devices generate IRQs. This behaviour can be controlled via the TrustZone Interrupt Controller [10], which like with the TrustZone Protection Controller, can only be accessed while executing in the secure world.

2.4 Related Work

2.4.1 GlobalPlatform

> "GlobalPlatform is a cross industry, non-profit association which identifies, develops and publishes specifications that promote the secure and interoperable deployment and management of multiple applications on secure chip technology. Its proven technical specifications, which focus on the secure element (SE), trusted execution environment (TEE) and system messaging, provide the tools that are regarded as the international industry standard for building a trusted end-to-end solution which serves multiple actors and supports several business models."[12]

GlobalPlatform provides security specifications for both hardware and software with the purpose of bringing interoperability and future-proofing. In this thesis a subset of the TEE API specification will be implemented.

The TEE API specification [13] consists of two parts, the Client API and the Internal API. It describes what methods should exist, what parameters they require and defines a number of requirements for the behaviour of these methods; what they must do, what they may do and what they can not do. The specification also defines a number of constants and structures used as parameters and return values of its methods.

The Client API [14] describes functions that can for example be used by REE Applications to open sessions with trusted applications, invoke commands on a trusted application or register shared memory to be used by the trusted application.

The Internal API [15] describes functions that can for example be used to perform encryption, signing and hashing. It also describes functions that assist with storing encryption keys and sensitive data. It lets the implementation define a number of supported encryption algorithms and includes arithmetic functions to enable customized implementations of encryption algorithms within a trusted application.

2.4.2 QEMU

QEMU [16], short for Quick Emulator, is an open source emulator and virtualizer. It can emulate many different platforms and it can be run as a native virtual machine. QEMU also powers the Android emulator, a part of the Android SDK, which makes it interesting for our purposes. QEMU is able to interface with many of the physical devices of the host and it is possible to use disk images to emulate usb or hard disk drives. At the time of writing, there is no support for TrustZone in the release version QEMU.

2.4.3 QEMU-TrustZone

The area of open emulation of trustzone is not well explored. Some work exists however. We took a deep look at one project in particular, a modified version

of QEMU. This modified version of QEMU was developed with the purpose of adding experimental TrustZone support[17]. The work expands upon QEMU's MMU implementation and adds MMU modes for the not secure state of the ARM processor, leaving the original MMU implementation to control the secure state of the processor. Further it simulates memory access restrictions by implementing a simplified model of both the Address Space Controller [18] and the TrustZone Protection Controller [11]. The interaction between normal and secure world is also implemented to enable entry to a secure monitor.

2.4.4 SafeG

SafeG [9] is a dual-OS monitor for TrustZone which focuses on optimizing real time embedded systems. It is designed to execute an RTOS(Real-Time Operating System) and a GPOS (General-Purpose Operating System) on the same hardware platform [19]. SafeG allows many different OSes to be run in either of the two worlds in TrustZone. The version of SafeG used for this work [20] is lightweight compared to the later releases but provides all the basic functionality required for our goals. It has the ability to load the two OSes, receive SMC and perform context switches. It also provides a number of system calls to this end. The QEMU release of SafeG is a major component of this work as everything done during this project is built with this release as the base.

2.4.5 FMP

In this work we will use the TOPPERS/FMP kernel [21] as the OS of the secure world of TrustZone. FMP is based on the specifications of uITRON4.0 [22] which, instead of processes, defines tasks that can be run periodically or continuously and be assigned different priority levels. These tasks are compiled with the OS. It is very lightweight and provides next to none of the standard c libraries (as is typical for most OS:es run in the Secure World). FMP comes bundled with the QEMU release of SafeG.

2.4.6 GNU Debugger

The GNU Debugger [23], or GDB for short, is a powerful debugger used as the standard debugger in the GNU operating system. GDB is able to remotely debug a system, a feature often used when debugging embedded systems. This work will add the possibility to debug trusted applications within the emulator using GDB.

Chapter 3
Approach

3.1 Introduction

At the start of this project; it was unclear what had already been done on the subject except that a version of QEMU modified for TrustZone existed, mentioned in section 2.4.3. Much of the preliminary work was dedicated to researching what functionality had been implemented as well as finding out what other components (monitor, OS for the Secure World) could be integrated. Finding the SafeG-FMP-QEMU release, getting it to build on our 64-bit machines, testing it and realizing that this was something that could be built upon was what really set off this project.

The modified QEMU, the SafeG monitor and the realtime kernel FMP all fit our needs well. Using these components as a base we have created a development environment where it is possible to test and debug trusted applications and security aware rich-os applications. This was achieved by finding out how to use the already integrated above mentioned components, implementing the global platform API and adding a communication protocol between the two worlds. The remainder of this chapter will first describe the general structure of the implementation and after that describe how the work was carried out step by step.

3.2 Structure of implementation

3.2.1 Starting Point

The groundwork used for this project was done before the start of this thesis, the starting point of this work can be found in the QEMU release of SafeG [20].

The hardware layer is emulated by a modified version of QEMU [24]. U-Boot [25] is used as the bootloader for the system and loads the SafeG monitor. The monitor in turn loads the kernel of both worlds and starts the trusted side, FMP. Once FMP is running it will yield execution and allow the normal world, running Linux in this implementation, to start. The normal world will run until the secure world has any task to perform. After the secure world has no current tasks it once again yields execution to the normal world.

The trusted side has priority over the untrusted side and will only yield execution to the untrusted side when it has no task running. This is done with an idle task that runs whenever nothing else is running, which gives execution time to the non-trusted side. This is a feature inherited from the FMP kernel used in the trusted world.

3.2.2 Adding Global Platform Support

With an emulated hardware platform in place we implemented the Global Platform Client API with functions that can be called by client applications in the normal world and the Internal API which is used by the trusted applications running inside the secure world.

In order to make use of the Global Platform API we needed a way to send data between the two worlds. This was done by constructing a communication protocol which routes commands and memory addresses from the Client API to the Internal API. For this to work we also needed to implement shared memory in order to enable the sending of larger chunks of data between worlds.

Global Platform API calls functions as follows in our implementation; Applications in the Normal world can make calls to the Global Platform Client API. These calls are handled and relayed to the communication protocol which sends the command and its data through the Normal World OS. Via the Secure monitor the data is then sent to the Trusted OS which delegates the command to the appropriate trusted application.

Trusted applications can use data written by the client application to shared memory and call on functions in the Internal API to perform secure operations such as encryption. Once finished the TA can write results, which are to be returned to the client application, to shared memory. Figure 3.1 illustrates the structure of the development environment.

3.3 Hardware Emulation

3.3.1 Introduction

QEMU is used for the underlying hardware emulation layer. The version of QEMU used in this work is a modified version which can found here [24]. Some modifications had to be made in order to realize our goals.

3.3.2 Secure Monitor Call

When running the rich-OS on this TrustZone enabled version of QEMU we found that attempts to execute the Secure Monitor Call, SMC, the instruction used to enter the monitor, resulted in an "Illegal instruction" error. By examining the source code we found that this occurred because the SMC instruction is not allowed to be called from user mode. This behavior is correct according to the specifications as SMC may only be called when in supervisor mode, see section 2.3.5, commonly known as "having root access". Supervisor mode should be gained by making a Supervisor Call, SVC [26]. Entering supervisor mode did not work as expected

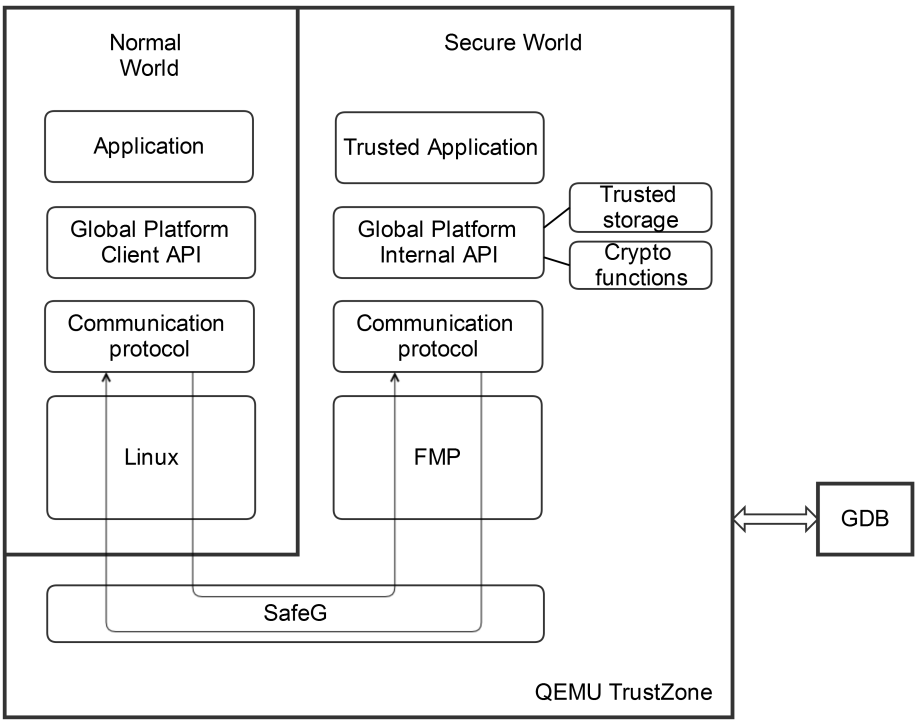

Figure 3.1: The structure of the development environment.

and from what we can tell, this QEMU version did not have support for the SVC instruction.

We decided that having the distinction between user mode and supervisor mode for the purpose of invoking SMC was unnecessary for our intentions and thus implemented a work-around [27]. This was done by simply making QEMU disregard the operation mode when an SMC call was invoked. The reasoning for this is that for any attack-scenario, the user should always be considered to have root-access. As the main goal of this work is to provide a development environment for client and trusted applications using Global Platform, we believe that this alteration would have no impact on its usability.

3.3.3 Shared Memory and the Memory Management Unit

In order to transfer data between worlds we needed shared memory. Shared memory is memory that can be read and written by both the normal- and the secure worlds. Normally, each world has its own memory management unit, or MMU. To simulate this, the MMU implemented in QEMU uses separate look-up tables for look-ups coming from the normal world and look-ups coming from the trusted world. This meant that the virtual address sent through to the trusted-side resolved to a different physical address in the two worlds. Upon researching how this problem is generally solved on current platforms we found that much information on the actual implementations is classified. From what we could gather when consulting our supervisors, a common approach is that some section of the memory is defined as shared and then treated differently by the MMU.

Initially we considered this approach, but this solution came with several problems such as having to rewrite the dynamic memory allocation on the rich-os side in order to make it possible to choose where memory should be reserved.

Another solution was close at hand however, as there is no set definition on exactly how shared memory should be implemented. There would also be no security concerns with letting the trusted world have full access to all hardware. Since we have full control of the hardware (QEMU), the monitor and all the software, we have full knowledge of the structure of the memory addresses. The Secure World's memory stack starts at address 0x81000000 and grows (upwards) from there while the Normal World uses the address space 0x0-0x60000000. Since the Normal World runs Linux this means that the stack will start at 0x60000000 and grow downwards while dynamic memory allocation starts at 0x0 and grows upwards. Using this knowledge, the look-up problem was solved by including a small condition in the trusted virtual processor's MMU:

If the translation request is being made from the secure side (the NS bit is set to 0) and the virtual address is lower than 0x80000000 the address should be resolved using the normal world's look up table.

This works very well because the only time that the secure side will use an address below 0x80000000 is if it has been handed a pointer to something that has been dynamically allocated in the normal world. The change also does not affect security in any way because letting the trusted world have full access to all

memory is within the specifications. Any call from the normal world is unaffected by the change. This modification gave the system crude but fully functional access to shared memory. The trusted world could access any memory that had been dynamically allocated by the normal world.

3.4 The Secure Monitor

SafeG is meant to be the monitor of a real-time system and as such, does not have a simple method of sending data between worlds. Instead it relies on a mailbox type communication module which was not finished and in a working condition for the version used in this thesis.

Conversation with the author of SafeG, Daniel Sangorrin Lopez, revealed that a new version of SafeG was soon to be released which would include english documentation as well as enhanced communication features. These enhanced communication features, primarily made for real-time systems, was not needed for this project however.

Because QEMU is at this point emulating a single-core system with no possibility of a context switch during execution in the secure world, code execution is very straight-forward. Making a switch to the Secure World is in this situation analogous to any other function call; it does not return until it has finished executing and a switch back to the normal world is invoked. If this function would accept a pointer to some struct defined by us it would be possible to implement any kind of call to the Secure World.

By analyzing the source code of SafeG we found that functionality to send an integer to the other world upon a switch existed. This feature was however not working as expected and was not fully implemented. This might be the reason it was left out of SafeG's documentation.

We modified the code so that it was possible to send integers between worlds. We found that when a system call invoking an SMC from the normal world was made the monitor would write two values, one of which was a status flag, into a return-arguments structure in register 1. In the secure world we declared a pointer to such a structure and wrote it to r1 when switching to the normal world. The monitor stores this pointer upon switching to the normal world and restores it when returning to the secure world. Once an SMC from the normal world was made, the monitor would populate the return argument structure with a single integer from the normal world.

3.5 Operating Systems Layer

3.5.1 Linux

Linux 3.6.10 is used as the kernel for the REE. For this work we make use of a minimal Linux kernel compiled for ARM processors. No modification had to be made.

3.5.2 FMP

FMP is used as the kernel for the TEE. The FMP kernel fit our needs well and was bundled into the SafeG release used for this project. SafeG was already configured to use FMP and as such not much time was needed to integrate it into the project.

Instead of processes, FMP defines tasks that are run by the kernel in order of priority. There are no services or background processes which you would find in a more full-fledged OS; if no task is executing then nothing is. There is also no file-system or possibility of loading a program (process) after compilation. All tasks are defined in a special configuration file which FMP reads to determine their properties as well as when and how they should be run and are then compiled into the kernel.

For the functionality Global Platform API defines however, there is no need for concurring tasks or special priorities. Because the Secure World cannot be interrupted by the Normal World during execution, there is no possibility of a new API call being made before the current one finishes.

The only task defined in our implementation is the "busy-wait task" which is continuously run with the lowest priority. Its original function is simply to yield execution to the Normal World after the other, higher priority tasks had finished executing. It achieves this by executing a system call to the monitor which invokes a context switch. Without this task, the Normal World would never be allowed to execute. When this is the only task running in FMP, FMP will only execute when a "Switch to Secure World" system call from the Normal World has been made ie., when the Normal World explicitly invokes a context switch through the monitor. Inside this task is where FMP can extract the pointer which was sent by the Normal World and dispatch it to our communication protocol. After it has been propagated through all layers and a return value has been set, the task will switch back to Normal World and wait for a new "Switch to Secure World" system call. Below is pseudo-code describing the modified "busy-wait-task".

```
"busy-wait-task" {
    struct return_args
    Packet *p
    while{
        switch_to_normal_world(&return_args)
        < normal world executes (for as long as it likes) >
        < normal world switches to secure world >
        p = ret_args.arg1
        send p to our communication stack
    }
}
```

3.6 Communication Protocol

In order to send data between the two worlds we needed some sort of communication protocol. By using the knowledge found when examining the monitor we began work on a protocol which would let us send both commands and data from

the normal world to the secure world, and for the secure world to return processed data.

With the SafeG monitor it was possible to send a single integer value to the other world via piggybacking when performing a switch between worlds. This way messages could be sent between worlds. Using this basic functionality to append a single integer, communication was expanded by sending the address of a packet, a structure containing information to be passed to the other world.

The sent packet contains an ID-value to determine what type of command should be executed and a pointer to another structure containing the necessary data for the command to be performed. Finally it contains a pointer that can be used by the trusted environment to write the result of the action.

Beyond the integer sent to the monitor, all data, packets and structures are written into shared memory. An application in the normal world allocates memory needed for its commands and data to be sent, as well as memory needed for the return data. The allocated memory is then registered as shared memory.

The communication developed always sends requests from the normal world to the trusted world. Communication can be one directional this way because the trusted world will never send requests to the normal world. Because the trusted world has absolute priority over the normal world, once the request is sent, the normal world is not allowed to run until the trusted world yields which means that in practice, a function that calls the trusted world will not return until the trusted world is finished processing the request.

3.7 Implementing the GlobalPlatform API

3.7.1 Introduction

We implemented the Client API and sections of the Internal API. The TA Interface as well as all structures and constants defined in the API are critical to implement in order to achieve any form of functionality. Memory management, trusted storage and cryptographic functions were chosen because they contain functions most likely to be required by a TA.

3.7.2 Dynamic memory allocation

The GlobalPlatform defines a couple of functions analogous to malloc, realloc, free etc found in the standard library (stdlib.h) you would normally use for C programming. Because FMP is so lightweight it offers no support for this standard library. It also does not have its own dynamic memory handler.

Implementing dynamic memory allocation is not a complicated task. Several examples of this can be found with ease [28]. In order to implement dynamic memory the functions sbrk() and brk() are needed; however these do not exist in FMP either. The use of sbrk and brk is as follows:

"brk() and sbrk() change the location of the program break, which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has

the effect of allocating memory to the process; decreasing the break deallocates memory." [29]

Trying to access or allocate memory without moving the program break normally results in a segmentation fault. FMP however does not have any specific memory allocated to a specific process. Instead the entire secure world could be referred to as a single process with full access to all memory addresses. We chose to define a certain area of memory addresses (currently 0x91000000-0x923FFA10) as the heap and implement brk() and sbrk() on this area. Using this newly defined heap, we implemented malloc, free and realloc using the guidance found at [28].

3.7.3 TA Interface

The internal API defines an interface which all trusted applications must implement in order to receive calls from the client. The functions, or entry points, defined in the interface are: create, destroy, open session, invoke command and close session and they all have corresponding functions in the Client API. In the API specification documents, the structure of the input parameters for the entry points differ from the corresponding function calls in the Client API. This means that the data has to be converted to match the internal API; this is handled by our communication protocol.

A typical usage of the entry points is create->open session->close session->destroy. If several commands needs to be sent to the TA or the TA is expected to be reused, the invoke command entry point can be used after the session has been opened and may be used as many times as needed before close session is called. If the TA is only meant to do one specific thing one time, it is possible to send the required data with the open session call directly. With that said, the specifics regarding how any one TA chooses to use its entry points are not defined in the API; the developer of the client application must still know how to use a specific TA in order to actually make use of it.

3.7.4 Trusted Storage for Data and Keys

Trusted Storage is centered around objects. The Global Platform API defines two categories of objects, persistent and transient. Both can be either an encryption key or a key-pair. Persistent objects may in addition contain data or can alternatively be pure data objects. As their names imply, transient objects are temporary and remain only until closed or the TA instance that created them is destroyed. Persistent objects remain until they are explicitly deleted. Each trusted application has access to a set of Trusted Storage Spaces which are shared among all instances of the TA. Version 1.0 of the specification only defines a single storage space for each TA [15].

Each cryptographic object can be one of many types. The type is dependant on the key it contains. The object has a list of attributes which depend on the type of the object. An AES key object contains only a "secret value" while an RSA-public key object contains the attributes "RSA modulus" and "RSA public exponent".

In this implementation each object contain an array of all possible attributes. Each element is used, and labeled whether it is used or not, depending on the type of the key object. While not optimal from a memory usage perspective (most listed attributes are never used for a given object) this structure is a simple initial implementation providing adequate functionality. The overhead created was deemed small enough not to matter when running tests within this environment.

FMP is very minimal and does not have a file system implemented. In our implementation persistent storage is implemented in dynamic memory and only persists while the TEE is running, whenever the system is restarted the persistent storage is therefore wiped. This was done as a simple solution to test other functionality as soon as possible. Implementing a fully persistent storage has yet to be done as it was deemed to be of low priority for this work.

3.7.5 Cryptography functions

We decided to include the most commonly used cryptographic functions. This was done with the reasoning that it would improve the usefulness of the development environment greatly compared to just having stubs for all crypto functionality. The current version of our development environment has the following cryptographic functions implemented:

- Hash functions
  - SHA-1
  - SHA-256
- Symmetric crypto
  - AES ECB
  - AES CBC
  - AES CTR
- HMAC
  - SHA-1
  - SHA-256

The implementations of the hash functions and the symmetric crypto functions are largely extracted from the libtomcrypt [30] library while the HMAC is our own and done based on RFC 2104 [31]. These crypto functions were chosen because they are among the ones most commonly used in the business and they are sufficient to fulfill most requirements.

In order to use the implementations from libtomcrypt a lot of work was put into understanding the structure of that library. We also had to write our own porting layer in order to integrate it with the rest of our code and to adapt it to the way the GlobalPlatform API's crypto functions were designed.

3.8 Debugging

3.8.1 Adding GDB support

In order for GDB to remotely debug another system, which is what we are doing, that system needs to implement a GDB-stub. A GDB-stub can be viewed as a receiver that implements and exposes the GDB communication protocol to the outside. Conveniently, QEMU already has a GDB stub implemented. The compile option "-g" [32] enables extra debugging information which can be used by GDB. Because the entire trusted side, OS along with API and trusted applications, are all compiled into the kernel we can use the -g option during compilation and then load GDB with the compiled executable. Since we are using remote debugging, GDB itself will not run the code but instead read its symbol table.

In order to hook up GDB into QEMU, QEMU needs to be started with the options "-s -S", which makes it stop and wait for a GDB connection before it executes any code. After loading the compiled FMP kernel into GDB you use the command "target remote localhost:1234" to connect it to QEMU's GDB stub; from there on it is possible to use GDB as usual to debug. Currently, it is only possible to debug one side at a time. We integrated the GDB loading procedures into our run-script in order to automate this process for the purpose of debugging trusted applications and the secure world. It is however possible to load a program on the non-secure world into GDB instead and use it in a similar manner.

Chapter 4
Results and Discussion

4.1 Introduction

Before this work started, there was no easy way for a developer to start working with TrustZone. Emulators that existed before either required a license or were difficult to begin using. Buying hardware to develop on is costly and cumbersome. Using our work, applications and TAs utilizing the Global Platform API can be run and tested in an easy and open-source manner.

4.2 Modifications and Functionality Developed

A number of modifications had to be done to several of the components we used in addition to implementation of the Global Platform API:

- QEMU-TrustZone did not have a functional supervisor call to enter supervisor mode and thereby enabling the use of secure monitor call. We made a change to ignore the need to be in supervisor mode when performing a secure monitor call. The reasoning being that an attacker will always have root access, that is, be in supervisor mode.
- We implemented shared memory in QEMU-TrustZone by resolving any address belonging to the normal world using the normal world mmu even when executing in secure mode. The reasoning was that the only time the secure world will try to access these addresses is when a pointer to dynamically allocated memory had been passed from the normal world.
- We analysed the source of the monitor used, SafeG, to find a way to communicate easily between worlds. Based on the feature in SafeG to piggyback an address during a context switch, we built a communication protocol for use with the Global Platform API.
- We implemented all functionality required of the Global Platform Internal and Client API in order to test and run most client and trusted applications. This includes memory functions such as malloc, free and realloc which did not exist in the kernel used in this project as well as storage functions and crypto functions. All functions needed to communicate between client and trusted applications were also implemented.

- We researched what debugging possibilities existed for our work and integrated the Gnu Debugger into the development environment.
- We bundled everything in an easy to use package with several scripts (many of which are extended versions of what was already included in the SafeG-QEMU release[12]) to support installation and ease-of-use, well documented example code and instructions for how to set everything up and run it both on a real machine and in a virtual environment.

4.3 Summary of Functionality

Our release [33] enables anyone with access to a computer to develop trusted and non trusted applications that makes use of the Global Platform API and TrustZone functionalities. The installation process is mostly automated through scripts, instructions for how to start using everything is included and the code is documented. There are also many scripts for automated building and running. The release can be run on a physical machine or in a virtualbox. Operating System recommendations and instructions for how to run it in Oracle VM VirtualBox [34] are also included.

4.4 Testing the implementation

We developed several trusted applications and clients to test the implementation. Using our client applications it is possible to access the trusted applications in the secure world and encrypt as well as decrypt text messages (typed into a console) using one of several encryption algorithms. In a similar manner it is possible to hash messages using a number of different hashing algorithms.

A demo client application and TA providing above mentioned functionality (utilizing the Global Platform API) are included in the release. In addition a TA used to unit test both the API and many internal functions is included. Instructions for how to use these can be found in the documentation [33].

4.5 Walkthrough of a Usage Scenario

This will be a walkthrough of what happens on the API level on both sides during a typical use case scenario. The code-snippets have been truncated and cut in order to only showcase the essential parts; the full code is available in our release [33].

First, the client application initialize the context and opens a session with the TA:

```
[Client Side Code]
/* initialize context */
  result = TEEC_InitializeContext(NULL, &context);

  operation.paramTypes = TEEC_PARAM_TYPES(
```

```
TEEC_VALUE_INPUT,TEEC_NONE,TEEC_NONE,TEEC_NONE);
  operation.params[0].value.a = algorithm;

  /* open session with demo_trustlet */
  result = TEEC_OpenSession(&context,&session,
  &demo_trustlet_TEE_UUID,NULL,NULL,&operation,NULL);
```

Initialize context only confirms that TrustZone environment exists and is available. The call to TEEC_OpenSession however reaches the TA's corresponding function TA_OpenSessionEntryPoint:

```
[TA Side Code]
TEE_Result TA_EXPORT demo_trustlet_TA_OpenSessionEntryPoint(
uint32_t paramTypes, TEE_Param params[4], void** sessionContext ){

    int algorithm = params[0].value.a;
    switch(algorithm){
    case TEE_ALG_AES_CTR:
    result = TEE_AllocateOperation(
&demo_operationHandle, TEE_ALG_AES_CTR, TEE_MODE_ENCRYPT, 128);
    break;
    case ...
    }
```

In the TA's OpenSession function, we extract the algorithm that the client wishes to use and use the Global Platform API call TEE_AllocateOperation to allocate resources needed to perform it. The function eventually returns one of the return values defined in the API and that value is checked by the client. The client then need to register shared memory in order to send and receive data from the TA. This is done with the TEEC_RegisterSharedMemory function.

```
[Client Side Code]
inputSM.size = inputSize;
inputSM.flags = TEEC_MEM_INPUT;
inputSM.buffer = (uint8_t*)inputBuffer;

result = TEEC_RegisterSharedMemory(&context,&inputSM);
if (result != TEEC_SUCCESS)
{
printf("failed to register shared memory for inputbuffer");
}
```

When registering shared memory it is important to select whether it will be used for input, output or both. The next step for the client is to set which key to use for the operation, in this case, the key is taken from user input but it could also have been stored in the secure world.

```
[Client Side Code]
memcpy(inputSM.buffer, &key, strlen(&key));
operation.paramTypes = TEEC_PARAM_TYPES(
TEEC_MEMREF_PARTIAL_INPUT,TEEC_MEMREF_PARTIAL_INPUT,
TEEC_NONE,TEEC_NONE);

operation.params[0].memref.parent = &inputSM;
operation.params[0].memref.size = strlen(&key);

result = TEEC_InvokeCommand(
&session,demo_CMD_SET_KEY,&operation,NULL);
```

When TEEC_InvokeCommand is called, a context switch occurs and the TA for which the session was opened with receives the data in its corresponding function named TA_InvokeCommandEntryPoint. The parameter params contains all information sent by the Client; the operation to perform, the input data and where to write the output. As stated above, in this instance it needs to create a key object, populate it with a key and set it on the OperationHandle:

```
[TA Side Code]
TEE_Result TA_EXPORT demo_trustlet_TA_InvokeCommandEntryPoint(
void* sessionContext,uint32_t commandID, uint32_t paramTypes,
 TEE_Param params[4] ){

/* This is where all the crypto operations are performed.
*the params variable contains whatever the invoking
* program on the non-trust side sent us. */

result = TEE_AllocateTransientObject(
TEE_TYPE_AES, 128, &demo_aes_key);
/* declare and allocate the secret value attribute on the stack */

TEE_Attribute secret_value;
secret_value.content.ref.buffer = params[0].memref.buffer;
secret_value.content.ref.length = params[0].memref.size;
secret_value.attributeID = TEE_ATTR_SECRET_VALUE;

/* populate the key object with the secret value attribute.
* this will copy the buffers. */

result = TEE_PopulateTransientObject(
demo_aes_key, &secret_value, 1);
/* set the key object we just allocated and populated to
*  be used in our operation this copies all key material into
*  the operation and when this function returns
*  there is no connection between the operation
*  and our key object anymore*/
```

```
result = TEE_SetOperationKey
(demo_operationHandle, demo_aes_key);
char* IV = params[1].memref.buffer;
size_t IVLen = params[1].memref.size;

/* initialize the operation with the initialization vector
* (even if the operation doesn't need an IV we can still
* input one to this function. it just won't be used */

TEE_CipherInit(demo_operationHandle, IV, IVLen);

/* at this point the operation is ready to accept input
* data by use of the functions TEE_CipherUpdate
*  and TEE_CipherDoFinal */

return result;
```

After this, our client will fill the registered shared memory with the data it wishes to encrypt with AES. This data is taken from user input. The client will use the call TEEC_InvokeCommand again to order the TA to perform operations on the data:

```
[Client Side Code]
operation.paramTypes = TEEC_PARAM_TYPES(
TEEC_MEMREF_PARTIAL_INPUT, TEEC_MEMREF_WHOLE,
TEEC_NONE,TEEC_NONE);

 operation.params[0].memref.parent = &inputSM;
 operation.params[0].memref.size = msgLen;

 // no need to specify size since the paramtype is memref_whole
 operation.params[1].memref.parent = &outputSM;

 totalMsgLen += msgLen;

result = TEEC_InvokeCommand(
&session,demo_CMD_ENCRYPT,&operation,NULL);
if(result != TEEC_SUCCESS){
printf("failed to do digest. error code: %d", result);
 }
```

Upon receiving this command, the TA uses the TEE_CipherUpdate function which adds data to the ongoing cipher operation and performs an encryption or a decryption if it has enough input.

```
[TA Side Code]
result = TEE_CipherUpdate(demo_operationHandle,
params[0].memref.buffer, params[0].memref.size,
params[1].memref.buffer, &params[1].memref.size);
```

4.6 Limitations

4.6.1 Idle scheduling

When the TEE is running, the REE is not able to run. Execution of the REE is resumed once the TEE finishes. This has made the implementation of the Global Platform API simpler with regards to return-values and waits. In a 'real' system, there are no guarantees that the TEE won't be interrupted but this should not be a problem as long as the programmer uses the API as intended. It should be noted that there is currently no possibility for mixed priority for processes or multi-threaded execution across the TEE and REE.

4.6.2 Monolith TEE and Trusted Applications

Trusted applications have to be compiled together with the trusted OS and are run together as one large application. This forces users to recompile the trusted OS every time a change has been made in any trusted application. Because all trusted applications and the trusted OS share the same process on the trusted side and is compiled together, extra caution must be taken when naming global variables and functions in order to avoid collisions. This also means that all trusted applications are exposed to the whole internal API in addition to other trusted applications' functions. The programmer must make sure to only call functions declared in the Global Platform API specifications, typically prefixed by "TEE_".

4.6.3 Functionality

We have successfully added what we set out to do for this thesis, but this emulator is not feature complete. Simple applications can be run and tested but more complex applications may be outside the scope of the Global Platform API implemented. Persistent objects are not stored between system startups and require some considerations when testing applications making use of this type of objects. Especially critical is the fact that sharing rules for persistent objects are not enforced. This means that in practice it is possible to open several instances of the same object despite not using the share flag, which might lead to undefined behaviour if the persistent object subsequently is deleted.

Chapter 5
Conclusions and Future Work

5.1 Conclusions

The goal of this thesis was to enable testing and development of trusted applications using the global platform API within an emulator. We have shown that using already existing software it is possible to create a testing environment for trusted applications and clients. This was done using a modified version of QEMU with experimental TrustZone support and a monitor developed with real-time systems in mind. A subset of the Global Platform API was implemented for the purpose of testing applications making use of the API. We wrote trusted applications and clients that could take advantage of the TrustZone architecture through the GlobalPlatform API. With the inclusion of GDB it is possible to monitor the state of trusted applications within the emulator and to monitor the state of the emulated hardware. We hope that our work will be useful to anyone developing applications implementing the open specifications by GlobalPlatform.

5.2 Future Work

Our prototype is not yet feature complete. To be fully functional the missing functions specified in the Global Platform API needs to be implemented. Encryption and trusted storage was prioritised in this work because of their importance in trusted applications. We have with our work built a starting point and have commented the code as to make further development as easy as possible.

It would also be interesting to incorporate our work into the Android emulator. Currently, the Android emulator is based on an old version of QEMU which has been heavily modified to support the android OS and GUI. This means that given enough understanding of what modifications that has been made and why, it should either be possible to integrate the TrustZone modifications into the Android emulator or the emulator modifications into the newest version of QEMU with TrustZone support. Our limited research into this subject found that it would probably not be trivial however.

References

[1] GlobalPlatform Media Guide, [Online] Available from:

http://www.globalplatform.org/mediaguideSE.asp

[2] Trusted Platform Module Specification, [Online] Available from:

http://www.trustedcomputinggroup.org/
resources/tpm_main_specification

[3] VMware vSphere Hypervisor, [Online] Available from:

http://www.vmware.com/se/
products/vsphere-hypervisor

[4] The Xen Project Hypervisor, [Online] Available from:

http://www.xenproject.org/developers/
teams/hypervisor.html

[5] Forbes article, 2013, As Gadgets Shrink, ARM Still Reigns As Processor King, Last Accessed 2014 May 22 [Online] Available from:

http://www.forbes.com/sites/parmyolson/2013/
09/20/as-gadgets-shrink-arm-still-reigns-as-processor-king/1

[6] ARM TrustZone Description, [Online] Available from:

http://www.arm.com/products/processors/
technologies/trustzone/index.php

[7] Cortex-A9 Revision: r4p1 Technical Reference Manual 4.2, [Online] Available from:

http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388i/
DDI0388I_cortex_a9_r4p1_trm.pdf

[8] ARM Security Technology - Building a Secure System using TrustZone Technology, Section 3.2.1 The AMBA3 AXI system bus, [Online] Available from:

http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf

[9] Daniel Sangorrin Lopez, Advanced integration techniques for highly reliable dual-os embedded systems, Embedded and Real-Time Systems Laboratory Nagoya University (Japan), July 27 2012

[10] AMBA^{TM}3 TrustZone Interrupt Controller (SP890) Technical Overview, [Online] Available from:

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dto0013b/BEIJGAAH.html

[11] PrimeCell Infrastructure AMBA3 TrustZone Protection Controller (BP147) Technical Overview Revision: r0p0, [Online] Available from:

http://infocenter.arm.com/help/topic/com.arm.doc.dto0015a/DTO0015_primecell_infrastructure_amba3_tzpc_bp147_to.pdf

[12] About GlobalPlatform, [Online] Available from:

http://www.globalplatform.org/aboutus.asp

[13] GlobalPlatform Public API specifications, [Online] Available from:

http://www.globalplatform.org/specificationsdevice.asp

[14] TEE Client API Specification v 1.0 Public Release, Document Reference: GPD_SPE_007, July 2010

[15] GDT TEE Internal API Specification v1.0 Public Release, Document Reference: GPD_SPE_010, December 2011

[16] QEMU, [Online] Available from:

http://wiki.qemu.org

[17] Johannes Winter, Paul Wiegele, Martin Pirker, and Ronald Tögl, A Flexible Software Development and Emulation Framework for ARM TrustZone, INTRUST 2011, LNCS 7222, pp. 1-15, 2012.

[18] TrustZone Address Space Controller (TZC-380) Technical Reference Manual, [Online] Available from:

http://infocenter.arm.com/help/topic/com.arm.doc.ddi0431b/DDI0431B_tzasc_tzc380_r0p0_trm.pdf

[19] SafeG description and download, Last Accessed 2014 May 15, [Online] Available from:

http://www.toppers.jp/en/safeg.html

[20] SafeG for QEMU download, Last Accessed 2014 May 15, [Online] Available from:

http://support.toppers-open.org/safeg-qemu/

[21] FMP description and download, Last Accessed 2014 May 15, [Online] Available from:

http://www.toppers.jp/en/fmp-kernel.html

[22] uITRON Specification, [Online] Available from:

http://www.ertl.jp/ITRON/SPEC/mitron4-e.html

[23] Gnu Debugger, [Online] Available from:

http://www.sourceware.org/gdb/

[24] QEMU with TrustZone Support download, Last Accessed 2014 May 15, [Online] Available from:

https://github.com/jowinter/qemu-trustzone/tree/for-linaro

[25] U-Boot, [Online] Available from:

http://www.denx.de/wiki/U-Boot

[26] ARM SVC, Last Accessed 2014 May 15, [Online] Available from:

http://infocenter.arm.com/help/index.jsp?topic
=/com.arm.doc.dai0179b/ar01s02s07.html

[27] Cortex-A9 Revision: r4p1 Technical Reference Manual 4.2, [Online] Available from:

http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388i/
DDI0388I_cortex_a9_r4p1_trm.pdf

[28] Marwan Burelle, A Malloc Tutorial, Laboratoire Systeme et Securite de l'EPITA (LSE), 2009 February 16 , [Online] Available from:

http://www.inf.udec.cl/~leo/Malloc_tutorial.pdf

[29] Linux Programmer's Manual, brk() function, Last Accessed 2014 May 15, [Online] Available from:

http://man7.org/linux/man-pages/man2/brk.2.html

[30] LibTomCrypt, [Online] Available from:

http://libtom.org/

[31] RFC 2104, [Online] Available from:

http://tools.ietf.org/html/rfc2104

[32] Options for Debugging Your Program or GCC, Last Accessed 2014 May 15, [Online] Available from:

`http://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html#Debugging-Options`

[33] Release of the work described in this report, [Online] Available from:

`ftp://gooselab.net:65000/kraken_v1.0.tar.gz`

[34] VirtualBox, [Online] Available from:

`https://www.virtualbox.org/`

LUND
UNIVERSITY

http://www.eit.lth.se