TrustFrame, a Software Development Framework for TrustZone-enabled Hardware

João Pedro Cohen Rocheteau e Silva Ramos

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. João Nuno de Oliveira e Silva
Prof. Nuno Miguel Carvalho dos Santos

Examination Committee

Chairperson: Prof. Nuno João Neves Mamede
Supervisor: Prof. João Nuno de Oliveira e Silva
Member of the Committee: Prof. Ricardo Jorge Fernandes Chaves

November 2016

Acknowledgements

I would like to thank both my thesis advisors, Professor João Nuno Silva and Professor Nuno Santos, for the opportunity to carry out this work. Their availability and guidance, were essential for the course of this work. I would also like to thank Nuno Duarte from INESC-ID and Tiago Brito, for their support during the initial development phase of this work. I also want to thank Sileshi Demesie, for the crucial knowledge and support he provided in this phase of the work, to Instituto Superior Técnico (IST) and INESC-ID, for proving the means that allowed this work to accomplished, and to the Genode community, for their help during the entire thesis.

Last but not the least, I would leave a special thank you for all my friends and family, who gave me strength and support throughout this thesis.

Lisboa, November 2016
João Pedro Cohen Rocheteau e Silva Ramos

For my family,

Resumo

Com a contínua evolução de campos de tecnologia, como sistemas móveis, sistemas embebidos e computação ubíqua, a nossa forma de interagir com vários tipos de dispositivos está constantemente a mudar. Hoje, as aplicações com dados sensíveis são cada vez mais usadas nos dispositivos móveis dos utilizadores, o que traz grandes problemas de segurança. Numa tentativa de superar as preocupações de segurança crescentes, alguns fabricantes de hardware estão a começar a usar a tecnologia ARM TrustZone. Esta tecnologia apresenta-se com um enorme potencial de aplicação, uma vez que permite o desenvolvimento de sistemas operativos mais robustos, garantindo uma melhor segurança da aplicação. No entanto, investigação sobre este assunto é bastante complicada, devido à incompatibilidade de várias soluções de software e hardware, bem como a falta de documentação e apoio ao desenvolvimento de software para o hardware habilitado para TrustZone. Iniciar o desenvolvimento de qualquer solução baseada em TrustZone tem muitas barreiras: a selecção da ferramenta, a escolha de hardware compatível, configuração do ambiente inicial, estudo de APIs de programação e início do desenvolvimento. Neste trabalho estudamos o estado da arte actual em soluções baseadas em TrustZone. Este trabalho abre o caminho para o desenvolvimento de um quadro completo (documentação, ambiente de desenvolvimento e suporte de compatibilidade) para facilitar a inicialização de projetos de desenvolvimento de software baseado em TrustZone.

Abstract

With the continuous evolution of technology fields like mobile, embedded systems and ubiquitous computing, the way we interact with several types of devices is ever changing. Today, applications with sensitive data are increasingly used in mobile devices by users, which bring huge security issues. In an attempt to overcome the growing security concerns, some hardware manufacturers are starting to use ARM TrustZone technology. This technology presents itself with enormous appliance potential, since it allows the development of more robust operating systems, achieving better application security. However, research on this matter is quite complicated, due to various incompatible software and hardware solutions, as well as the lack of documentation and support to software development for TrustZone-enabled hardware. Development initiation on any TrustZone-based solution has many barriers: framework selection, choosing compatible hardware, initial environment configuration, programming APIs study and start of development. In this work we study the current state of the art in TrustZone-based solutions. This work paves the way for the development of a complete framework (documentation, development environment and compatibility support) to ease the bootstrap of TrustZone-based software development projects.

Palavras Chave
Keywords

Palavras Chave

Dispositivos Móveis

Segurança

Computação Confiável

ARM TrustZone

GlobalPlatform

Keywords

Mobile Devices

Security

Trusted Computing

ARM TrustZone

GlobalPlatform

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface

CA Client Application

CPSR Current Program Status Register

CPU Central Processing Unit

DMA Direct Memory Access

DoS Denial-Of-Service

DPM Data Processing Module

FIQ Fast Interrupt Request

I/O Input/Output

ICE Isolated Computing Environment

IDs Identifications

IRQ Interrupt Request

LOC Lines Of Code

MMU Memory Management Unit

NS Non-Secure

OS Operating System

OTP One-Time Password

REE Rich Execution Environment

RPC Remote Procedure Call

SCR Secure Configuration Register

SDK Software Development Kit

SMC Secure Monitor Call

SoC System-On-A-Chip

TA Trusted Application

TCB Trusted Computing Base

TDC Trusted Domain Controller

TEE Trusted Execution Environment

UUID Unique Universal Identifier

VM Virtual Machine

VMM Virtual Machine Monitor

1 Introduction

Mobile devices handle data that is becoming increasingly valuable and confidential. This data ranges from simple photos or files to our own medical information, banking credentials or even other access credentials, making them an increasingly sought target for a variety of attacks, as reported in [1, 2]. Furthermore, with the popularity of Internet of Things (IoT) rising, more and more devices are expected to be connected, exchanging these types of data. Therefore it is crucial to have mechanisms capable of protecting such data.

Nowadays, the processing of sensitive data is increasingly dependent on highly complex software, with many lines of code. Not only is this true for traditional computational platforms, like desktops and servers, but also for smartphones. Also, by depending on many lines of code, the operating systems may contain vulnerabilities that can be exploited by malware and possibly result in the theft or repudiation of data.

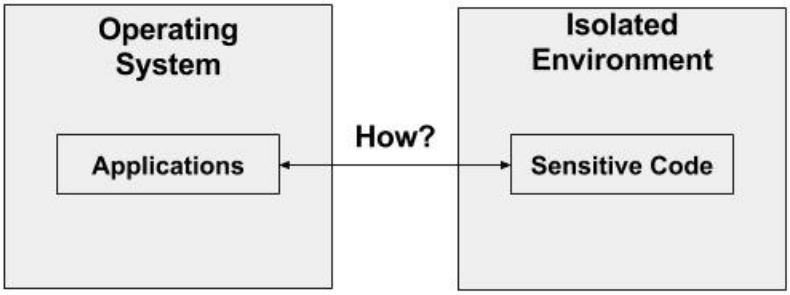

Figure 1.1: Isolation of sensitive code from the main operating system.

To develop secure and trusted software, new mobile platforms processors have been created that allow the execution of sensitive code, in a secure way, in particular ARM TrustZone [3] technology, present in the most recent ARM processors. It aims at enabling the creation of an execution environment, for protecting the confidentiality and integrity of critical code, allowing that code to be executed isolated from the main operating system (OS). Therefore, in the event of total OS compromise, the application remains secure. This allows most of the operating system and application code to be separated from the critical code, responsible for handling sensitive data or that executes in an isolated environment, creating a logical barrier between them. However, there are still difficulties in running sensitive code in isolated environments (see Figure 1.1). Samsung's KNOX [4] is an example of such an approach, as well as Samsung Pay[1], a mobile payment system that explores ARM TrustZone capabilities. More solutions with such capabilities will be discussed further on, in which the execution states, non-secure or secure, are managed by an hypervisor, or virtual machine monitor (VMM), responsible for creating

[1] http://www.samsung.com/us/samsung-pay/

and managing virtual machines. A platform, capable of offering such isolation for critical code, allows the construction of a panoply of security solutions.

1.1 Motivation

Despite the fact that ARM processors are widely available in smartphones, tablets, among other devices, ARM TrustZone functionality has not been fully explored. There are five main reasons for this.

TrustZone security mechanisms are complex. They suffer from compatibility issues between different hardware choices which require a great effort to overcome them, such as the creation of drivers for the target hardware, junction of APIs into a common API and common software development methods, to successfully minimize the impact of using different tools to achieve the same goal. Since TrustZone is rather used by the OS instead of being used by the applications themselves, although they enjoy the benefits of using TrustZone, optimizing the usage of resources by OSs is an important and ongoing research area.

Lack of good hardware documentation. Hardware manufacturers provide documentation on the hardware they sell, but sometimes it is not good enough. For developers with little or no experience, searching for answers in these documents may well be the same as trying to find a needle in a haystack. Some of these manufacturers also provide a community forum, allowing software developers to ask questions, but quite often, these questions do not receive the appropriate answer or do not get answered at all, which obliges the software developer to continue searching for solutions, wasting more time. As a consequence, these difficulties present themselves as a major challenge for the entire software development process.

Execution environment related difficulties. There already exist some frameworks capable of using ARM TrustZone technology, such as Genode OS Framework [5]. These development frameworks require the developer to initially configure them, setting the necessary toolchain, verifying the compile targets, and so on, before they can start using them. Again, for developers with little or no experience, just to install and configure a development framework can be a very difficult step, since the developer is not familiar to these types of environments and their requirements. However, they advertise their support of a certain development board and may require additional effort in configuring the environment, to use a hardware feature that may not be fully implemented already. These constraints bring even more difficulties, mainly due to the lack of good documentation for the framework being used. In this case, the developer is faced with the problem of having to explore both the hardware documentation and the framework documentation, which can easily be overwhelming for someone with little experience. Similarly to the case above, Genode provides some documentation, but some of it does not contemplate some probable issues that may appear for hardware other than the ones they mention.

Development board related difficulties. Every development board has its own software, that was designed to work for that specific board and not for any other. Althoug this is not an ARM TrustZone problem, it leads to more constraints for the developer in using the technology, since it sometimes requires different approaches to obtain the same desired outcome. This concern escalates if the developer introduces the selection of hardware, since there are multiple choices available that allow the use of

TrustZone and take advantage of its benefits. Each board may have its own way of doing the same step, such as the communication with the development host with proper software or manually, through the use of MicroSD cards, the difference in command targets and image mounting steps. These concerns lead the developer to consider aspects like compatibility between the board and the host. The availability of proper drivers that remove or diminish these constraints is not always assured.

Finally, **development boards evolution**. Some are now obsolete compared to new ones. The hardware has changed, introducing new hardware features and possibly new primitive instructions to be used. Also, their capabilities have been improved with better hardware, consequently being able to sustain more applications running, both in number and complexity, through the exploration of those added resources. This creates a gap between some of the first models used for this type of work and the new ones that are being released, that come already with changes, both in hardware and software, leaving the older models behind.

1.2 Goals and Requirements

The overall goal of this work is to conceive and implement a Software Development Kit (SDK) that eases the development process, as well as the testing of security applications for ARM TrustZone. The provided system should be compliant with the following requirements:

Ease the development process of new solutions. Instead of wasting time and resources on creating a custom platform that serves their intentions, the provided system should allow developers to jump straight to the development of their solution and start exploring the capabilities at their disposal.

Compatible implementation with API standards. Since the development of solutions for ARM TrustZone began, the interest of this type of research led entities, like GlobalPlatform, to develop specifications for projects on this matter. This non-profit organization has been developing specifications to facilitate the secure deployment and management of applications, in isolated environments. To keep the system current on this theme, this system should provide compatibility with these specifications.

Work on real hardware. This system should work on real development boards, where more and more developers may use and test it, in order to develop their own solutions. To allow such use and testing, this system will be used and tested for the NXP i.MX53 QSB board (formerly known as Freescale i.MX53 QSB) [6], since this is a regularly used development board for ARM TrustZone projects.

Efficient execution of applications. To allow developers to fully develop their idealized solutions, without having to sacrifice some of its features, the system should provide good scalability and usability. The execution of applications on top of it should present good overall performance and efficiency results.

It is intended that the system is not a final product, but instead serves as a basis on which future works might extend the current capabilities.

1.3 Contributions

The implementation of the proposed solution led to the development of a SDK, that allows the use of ARM TrustZone for newly developed applications. The main contributions resulting from this work are the following:

New SDK architecture. This work led to the creation of a new architecture, based on Genode, but using a custom Linux kernel which may be modified by developers, instead of a static prebuilt image, as in the original Genode architecture. Another aspect is a new layer on top of the system, which is composed by the GlobalPlatform API, according to the standards specified by GlobalPlatform.

Standardized API specifications on top of Genode. Currently, Genode does not support the API specifications from GlobalPlatform. We implemented that layer over our system, in order to support the API standards. This way, applications developed in other frameworks, compatible with the GlobalPlatform API specifications may invoke the same API, using our system, opening new possibilities for future development and testing of applications, already compliant with these standards.

Easy-to-use execution environment switch. During the development of the system, we needed to constantly change the execution environment. Since the base framework did not offer it, although internally it possessed its own switch mechanism, it was necessary to develop our own. This way, every developer can easily use our switch mechanism in their projects, without having to waste effort into developing their own from scratch. If the developer requires a modified version of our world switch, it already has a basis to work on, instead of having to fully implement one.

1.4 Structure of the Document

The rest of this document is organized as follows. Chapter 2 provides some background information on hardware features implemented in recent ARM processors, among other relevant topics addressed in this work. Chapter 3 provides an insight over the related work around systems that use TrustZone, followed by a summarized overview over existing development frameworks for ARM TrustZone. Chapter 4 describes the architecture of TrustFrame. Chapter 5 describes the implementation details of TrustFrame and Chapter 6 presents the evaluation method used to test our work, as well as the final results. Finally, Chapter 7 concludes this document by summarizing its main aspects, as well as some notes for future work.

2 Background

In this chapter we provide some necessary background information about the main topics covered in this work. First we explore ARM TrustZone, the technology used by our hardware (2.1). After that we describe the GlobalPlatform API, the API implemented on top of the system (2.2). Finally we explore the Linux Memory Management, a topic that contains useful information for this work, since this OS is used in the overall system (2.3).

2.1 ARM TrustZone

In this section we will explore the ARM TrustZone technology. In the following subsections we will address some details of this technology, in terms of its architecture, software development aspects and hardware requirements.

ARM TrustZone technology aims at providing a security framework that allows any device to sustain many of the threats that it will come across during its lifespan, like information disclosure, repudiation, among others. To do so, TrustZone technology gives SoC[1] creators the freedom, to select the specific components, that can fulfill specific functions within the security environment. Since its security extensions cover processor, memory and peripherals, as explained by ARM in [3], there is a variety of choices available. Prior to ARMv7 [7] and ARMv8 [8, 9] architectures, ARM TrustZone was not virtualizable, but with the added extensions brought by the new versions of the architecture, ARM introduced support for hardware virtualization, allowing the execution of separate operating systems, one secure and one non-secure, in separate virtual machines, managed by the hypervisor, a highly trusted management layer running in privileged mode.

2.1.1 Architecture

The partitioning of both hardware and software resources was the key element introduced, as referred in [3, 10], allowing them to coexist in two separate worlds, *normal world* and *secure world*, with independent memory address spaces for each processor mode, creating an isolated ambient that runs parallel to the device's operating system and exploring the concept of privileged and unprivileged mode already present in earlier ARM processors. The normal world or Rich Execution Environment, henceforth referred to as REE, is where all user applications are installed and execute, with limited access to the device's capabilities while the secure world or Trusted Execution Environment, henceforth referred

[1] System-on-a-chip is an integrated circuit with all the components of a computer or any other electronic system.

to as TEE, is where trusted applications run with full access to its resources (as illustrated in Figure 2.1). This isolation grants protection to trusted applications against installed user apps, running in the main operation system, by ensuring that no secure world resources are accessed by normal world components. This protection is assured through the hardware logic present in the TrustZone-enabled Advanced Microcontroller Bus Architecture (AMBA) bus fabric, which will be explained in detail further ahead. This protection also covers other trusted applications, since there is also isolation inside the TEE increasing, this way, the overall robustness of the whole environment, since both the REE OS and normal world applications do not have access privileges in the TEE.

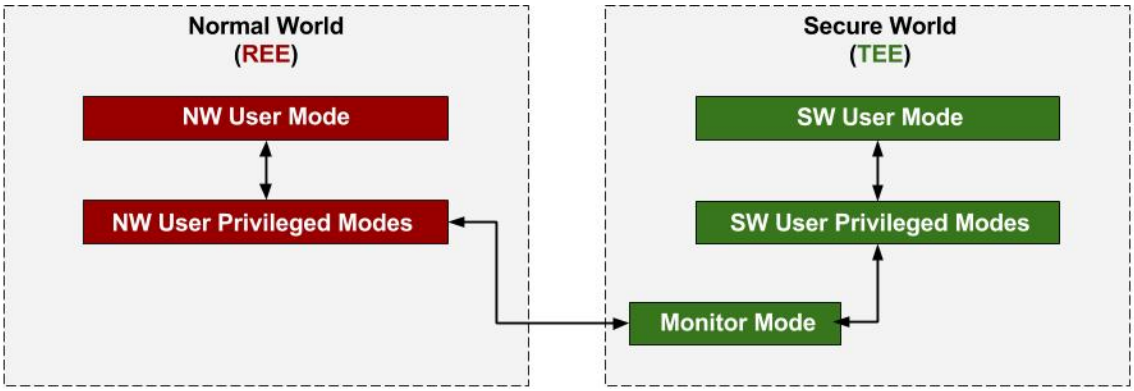

Figure 2.1: ARM processor modes with TrustZone security extensions.

As we head into more detail, we find several key components to the system's architecture. In terms of processor architecture, each physical processor cores provides two virtual cores and assigns each one to one of the different worlds, normal and secure. Between them lies a switch mechanism known as *monitor mode* (see Figure 2.1), that serves as a robust gatekeeper, in which both virtual cores switch context through time slices, saving their current state in separate virtual memory, so that the normal world cannot tamper with the secure world's memory. The memory regions used for both worlds can be hard-wired or configured. This configuration must partition the resources available, so that any non-secure access, to secure memory or device, is treated as a security violation, causing an external abort to the core. This configuration must always be done, regardless of the type of system or its requirements. When entering into monitor mode from normal world, tight verifications are required, for obvious security reasons, being only possible through specific exceptions like an interrupt, an external abort or via an explicit call to the Secure Monitor Call (SMC) instruction. The secure world entry to the monitor mode is less restrictive, since its coming from a core already executing in a secure environment, which is also the world in which monitor mode always executes, therefore being achieved by writing directly to the Current Program Status Register (CPSR)[2], along with the exception mechanisms present in the normal world-monitor mode exchange.

The execution world of the processor is determined by the value of the Non-Secure bit (NS bit) which, is present in the Secure Configuration Register (SCR), specifically in CP15[3] coprocessor. This bit works exactly like a flag: if it is set to 1, the virtual core that performed the instruction or data access

[2]Current Program Status Register is a 32-bit wide register used in the ARM architecture to record various pieces of information like execution mode and status flags.
[3]CP15 is the ARM processor system control coprocessor.

is executing in normal mode and can only access limited resources, instead of the full access granted to secure virtual processors (NS bit set to 0), for example, normal world software cannot access the contents of the SCR. This information is crucial to the Secure Monitor Mode handlers, since it ensures that no interrupts occur during the processing of an active FIQ interrupt. They determine whether to trap IRQ or FIQ interrupts, where the FIQ is a fast and high priority interrupt version of the IRQ, since it is prioritized by disabling both IRQ and other FIQ handlers during the request. This ensures that no interrupts occur during the processing of an active FIQ interrupt. A given FIQ interrupt will need to wait until the current FIQ interrupt is handled, only then will the processor core handle the new request. The same does not apply in case the core is currently handling an IRQ interrupt, since the IRQ does not disable FIQs, and so, the FIQ handler can interrupt the IRQ any time. If both interrupts occur at the same time, the core will handle the FIQ first and then the IRQ. Given these characteristics, IRQ interrupts are better suited as a normal world interrupt source and FIQ as the secure world source (as seen in Figure 2.2).

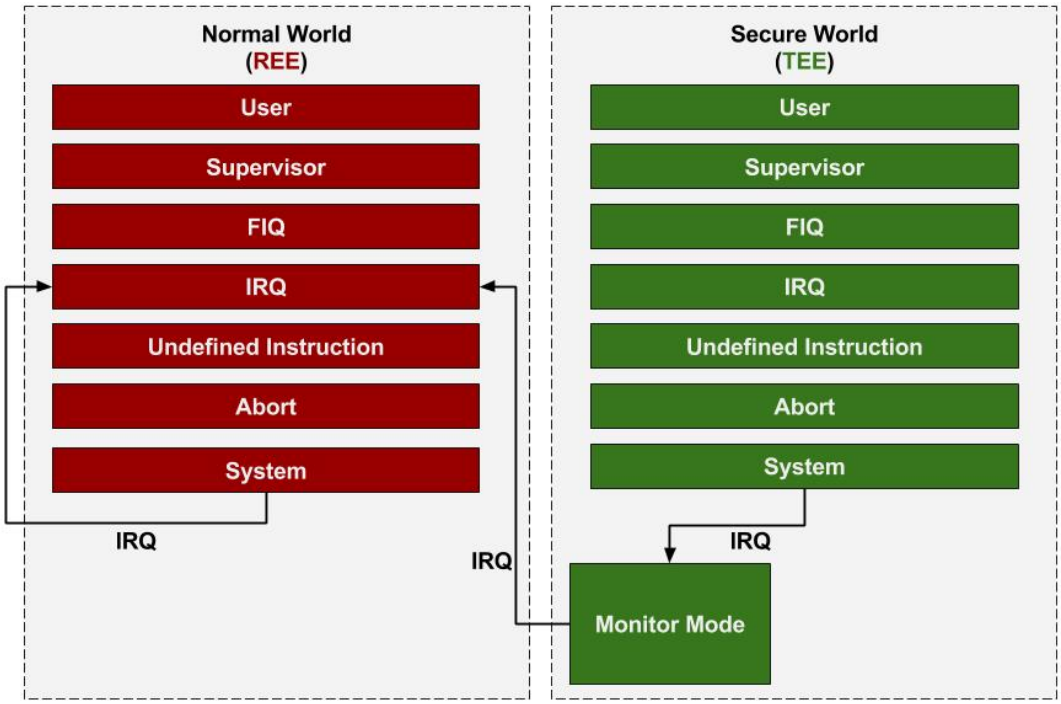

Figure 2.2: Secure interrupt design using IRQ as non-secure interrupt.

Outside of the processor core, the TrustZone architecture extensions also has the ability to both carry the NS bit in the AMBA3 AXI[4] bus through an extra signal, as well as to secure peripherals, like timers, interrupt controllers and user I/O devices, through the AMBA3 APB[5] peripheral bus. These capabilities makes them very important features, since it allows system designers to allocate resources exclusively to the secure world, instead of having to share them with normal world software and to choose which devices are allowed to be accessed by both worlds (see Figure 2.3).

[4] Advanced eXtensible Interface present in the Advanced Microcontroller Bus Architecture, which is the main SoC level system interface in the AMBA3 specification.

[5] Advanced Peripherals Bus interface, present in both AMBA2 and AMBA3 specifications.

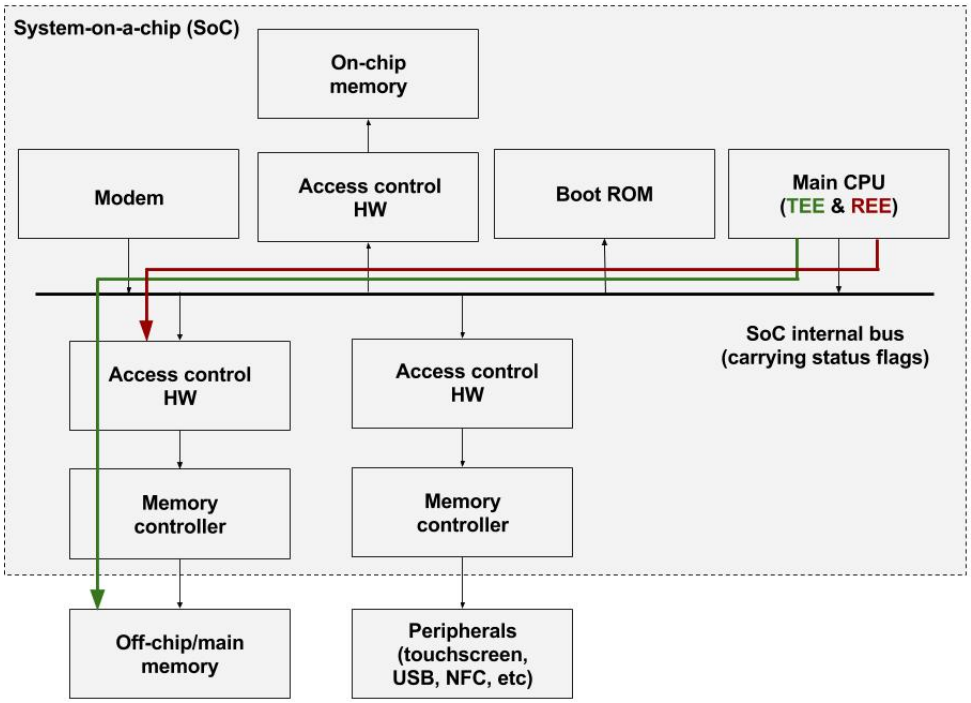

Figure 2.3: ARM TrustZone hardware architecture overview.

2.1.2 Software Development

With these hardware security extensions, OS designers have many possible software architecture approaches available, like the creation of an entire dedicated secure world operating system or a simple synchronous library of code placed in the secure world. These possible approaches allow the execution of multiple concurrent secure world applications and tasks, which run separately from the normal world environment. They take advantage of the user-space sandboxes created by the processor MMU[6] in the secure world memory space, allowing several security tasks to run at the same time, without trust issues. Both worlds do not share the same MMU, instead the hardware provides two virtual MMUs, one for each virtual core, allowing them to have local translation tables and full control over their virtual address to physical address mappings. The concurrency between applications may be restricted by the software implementation, since it can be simulated by the secure world OS, in order to further reduce the security risks associated with designing complex solutions. In case an access is attempted without the required permission, an external abort is sent to the processor, signaling a security violation. As for the synchronous library of code design, in some cases, having a simple library stored in the secure world, that can handle one task at a time, is enough for some applications. They only require software calls from the normal world operating system, making the secure world slave to the normal world in this case.

When implementing an entire secure world operating system, some security measures must be taken into account, like the boot sequence, where an attacker may tamper with the secure world oper-

[6]Memory Management Unit is the computer hardware component where all memory is referenced and translated from virtual memory address to physical memory addresses

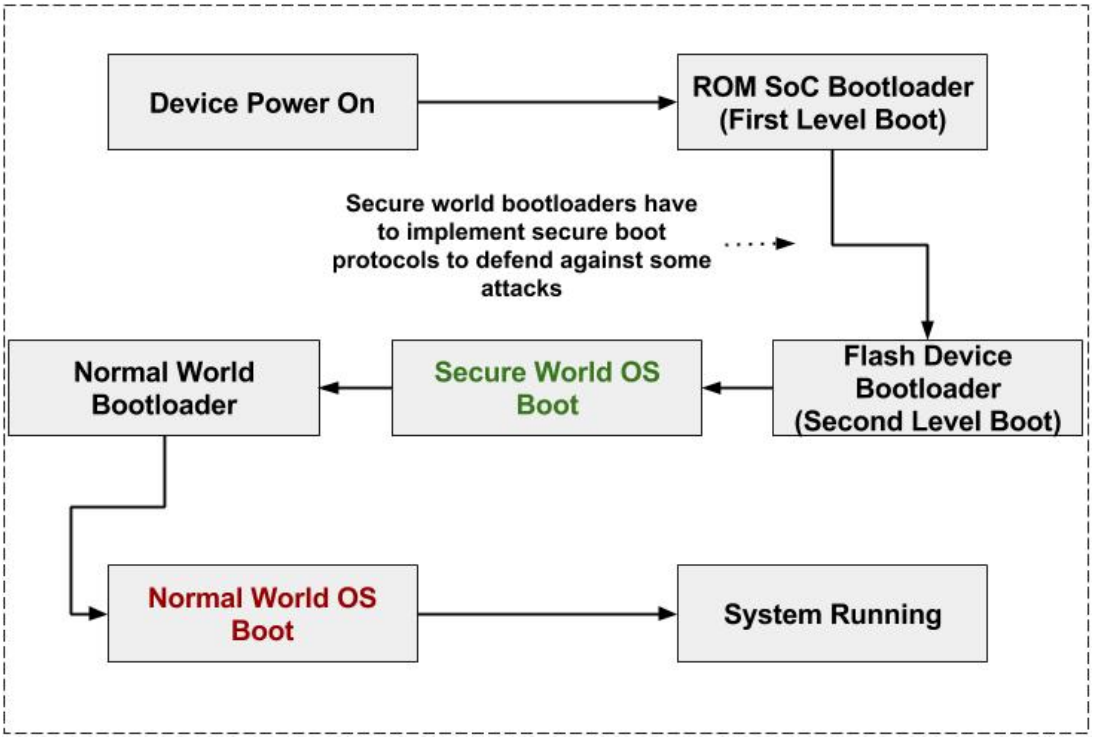

Figure 2.4: Boot sequence of a TrustZone-enabled processor.

ating system image. If the systems boots without validating the image stored in the flash memory, an attacker may replace that image for one that is vulnerable and with it, compromise the entire system. To avoid these constraints, every time a TrustZone-enabled processor boots, it starts in the secure world (see Figure 2.4). This enables the execution of security checks before the normal world is loaded, preventing system modifications by any existing software. The secure boot mechanism is composed of three pieces, according to ARM: Secure boot code located in on-SoC ROM, 256-bits of one-time password (OTP) fuse, which can contain a SHA256 hash of the RSA public key, owned by the Secure world software developer and a statistically unique secret key located in an on-SoC cryptographic accelerator. The combination of these pieces ensures a strong attack resistant mechanism.

2.1.3 Hardware Requirements

In order to properly develop software for TrustZone-enabled systems, one must have the appropriate tools, specially the necessary hardware. One slight constraint is that, sometimes, the hardware may have all the desired components to, theoretically, function accordingly, but may have the TrustZone capabilities disabled by default, i.e., restricted by the SoC manufacturer, which will require additional attention and intervention from the provider.

In table 2.1 we present some widely used development boards, that contain ARM TrustZone disabled by default. On the other hand, there are other development boards with TrustZone enabled, and that will serve for the task at hand (see Table 2.2). In this table we present some widely used development boards, that contain ARM TrustZone enabled by default. They also present some constraints, like

Board Model	Processor	Memory	Released	Price
Pandaboard	Cortex-A9	1GB DDR2	Oct 2010	$160-$190
Beagleboard	Cortex-A8	512MB DDR	Jul 2008	$95-$145
Beaglebone	Cortex-A8	256MB DDR2	Oct 2011	$89
Beaglebone Black	Cortex-A8	512MB DDR3	Apr 2013	$45-$90
Beagleboard-X15	Cortex-A15	2GB DDR3	Nov 2015	$199-$249

Table 2.1: Some development boards with TrustZone disabled.

the ones mentioned earlier, namely the exclusiveness of software tools for each board and the different approaches for the same end result. These constraints bring too many variables into play, since one must carefully choose the tools, both hardware and software, to work with, instead of simply choosing one option and going on from there, since this path may lead to a dead end.

Board Model	Processor	Memory	Released	Price
Raspberry Pi 2 Model B	Cortex-A7	1GB DDR2	Feb 2015	$40
NXP i.MX53 QSB	Cortex-A8	1GB DDR3	Jun 2010	$149-$299
NXP i.MX6	Cortex-A9	2GB DDR3	Jan 2011	$70-$129
ARM CoreTile Express	Cortex-A15	2GB DDR2	2011	$3000

Table 2.2: Some development boards with TrustZone enabled.

As for key components, strictly speaking, besides the TrustZone-enabled processor, every SoC manufacturer must also gather other important components [11]. Starting with a secure on-chip boot ROM containing the secure boot code, composed by cryptography code for image decryption and signature verification, that allows the boot in secure world. Followed by an on-chip non-volatile or one-time programmable memory for the device's secret keys, secure on-chip RAM, in which the trusted applications will be installed and run and where sensitive data will also be stored. Along with these core components, the device manufacturer will also need peripherals that allow trusted applications to gain access to their resources, in order to perform certain crucial tasks, like redirect touchscreen access to the secure world only, at a given time. In terms of available processors, currently ARM TrustZone technology is included in every Cortex-A series processor, ranging from the Cortex-A5 to the Cortex-A72, being the Cortex-A5, the low performance processor and the Cortex-A72 the high performance processor from the Cortex-A processor IP[7] lot. Therefore, today's development boards have a variety of processors at their disposal, based on the sort of tasks they are aiming to fulfill. New mobile devices, like the Samsung Galaxy S6, come with new and more powerful processors, like the ARM Cortex-A53, which is one of the top processors in the Cortex-A range supplied by ARM. Having better resources at their disposal, developers can create more robust and powerful secure apps, with more security features, consequently, increasing the overall security of the device.

[7] Intellectual Property is an invention or any labels created by an entity, in which the owner of the IP has the rights to gain from its creation.

2.2 GlobalPlatform API

To allow system developers to build security services for ARM TrustZone, ARM has initially provided her own TrustZone API called TZAPI. The API defined the interfaces in non-secure world, being publicly available. On the contrary, the API for the secure world was private and closed, leading to it not being accepted by the security industry. As a result GlobalPlatform [12], a non-profit, member driven association , saw an opportunity along with Trusted Computing Group [13], an association formed by AMD, HP, IBM, Microsoft and Intel, are working on creating a joined TEE standard, although GlobalPlatform has its own TEE standard since 2011.

Through the years, GlobalPlatform has employed a big effort to update their TEE specifications [14]. This API is divided into two parts, TEE Client API and TEE Internal API, for the normal world and secure world, respectively. The normal world application calls the TEE Client API, which then communicates with the secure world and sends relevant data. The secure world internally redirects this data to the desired trusted application, using the TEE Internal API. This API can be divided into five different API sets: Core, Secure Storage, Cryptographic Operations, Time and Arithmetic, as seen in Figure 2.5.

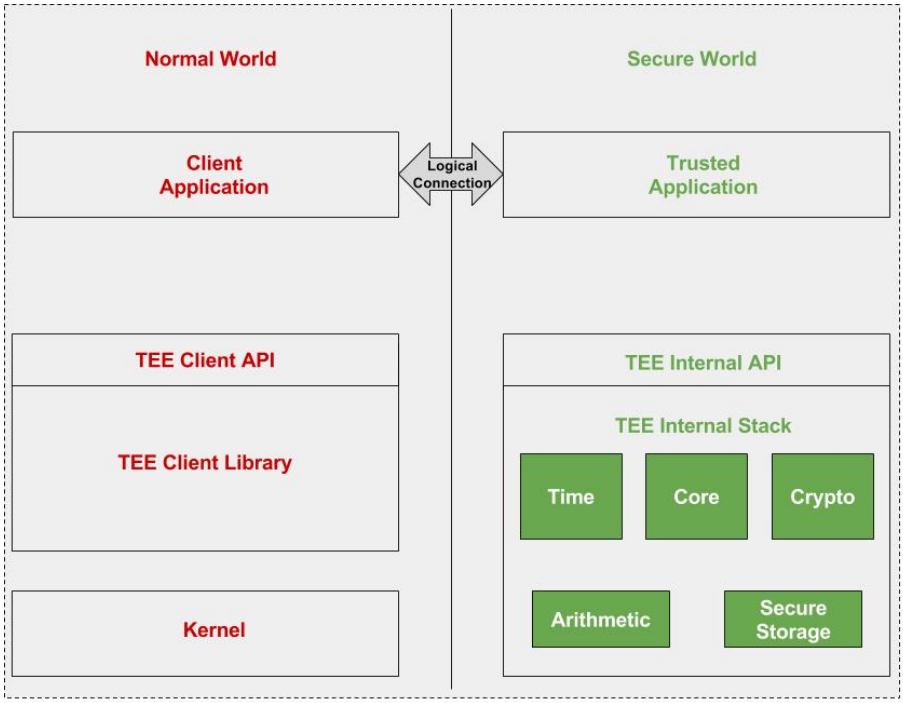

Figure 2.5: Software architecture for the GlobalPlatform API.

For a basic functionality between both world applications, the system must implement the core API, responsible for the operations over the received data and interactions between secure OS and trusted applications. The remaining sets are add-on specifications, meaning that although important and helpful, they are not required to be initially implemented. The Secure Storage API specifies an API that defines trusted storage for keys or general-purpose data. The Cryptographic Operations API specifies a set of cryptographic facilities for the generation and derivation of keys and support for several types of cryptographic algorithms, like AES, MAC and so on. The Time API specifies and API to access three

different sources of time: System Time, TA Persistent Time and REE Time. The System Time implies an arbitrary non-persistent origin and may use a secure dedicated hardware timer or be based on the REE timers. The TA Persistent Time is a real-time and persistent time, controlled individually by each trusted application (TA). The Arithmetical API serves the purpose of complementing the Cryptographic API, in case a Trusted Application needs to implement asymmetric algorithms, modes, or paddings not supported by the Cryptographic API.

Currently, its TEE Client API is in version 1.0 and its TEE Internal API is in version 1.1. These specifications are only guidelines for developers, they do not implement the entire API. It is expectable that future versions will have minor architecture changes compared to its current version, since these specifications are quite recent.

2.2.1 TEE Client API

The Trusted Execution Environment Client API [15] is the standardized API defined by GlobalPlatform[12], for the normal world, in order to allow communication between the client application (CA) and the trusted application (TA).

First, the client application has to connect to a TEE present in the secure world. This TEE can be one of several possible TEE, if the are more than one available to be used. The GlobalPlatform specifications cover this possibility, since the use of the API specifications imply compatibility with every TEE that implements the APIs. The differentiation between each TEE is given by a `TEEC_Context`, as defined in the API specifications, which is initialized upon invoking the `TEEC_InitializeContext` function. The `TEEC_Context` is the main logical container, establishing a link between a client application and a TEE. By doing so, the client application is able to specify which TEE it will connect and be easily adapted to the TEE available in a given system. Each TEE can have several trusted applications, protected from each other through both software and cryptographic isolation. They are then available to be invoked, by the client application, after a context has been initialized, with a given TEE. The bridge between the client application and the trusted application is materialized in a session, which is established by calling the `TEEC_OpenSession` function. The session is identified in the Client Application with a `TEEC_Session`. After the connection is established, the client can then give commands to the trusted application, by invoking the `TEEC_InvokeCommand` function. This function allows the client to specify which command is supposed to run in the secure world, over the parameters that are passed through shared memory mechanisms, being then able to receive an output response from the secure world trusted application. After the client application completes its desired commands, therefore terminating, it must close its session with the trusted application by invoking the `TEEC_CloseSession` function. In the end, to terminate the link between the client application and the TEE, the client application must invoke the `TEEC_FinalizeContext` function.

By following these specifications, it is guaranteed that all CAs will possibly have many similarities, in terms of its structure and source code, becoming familiar to developers, besides the compatibility guarantees with TEEs compliant with the GlobalPlatform TEE specifications.

2.2.2 TEE Internal API

The Trusted Execution Environment Internal API [16] is the standardized API defined by GlobalPlatform, for the secure world, in order to develop trusted applications (TAs) compliant with the requirements defined in the TEE Client API. These TAs are programs that run in a TEE and expose security services to the clients that connect with them, through client applications (CAs). Each TA is identified by a Universally Unique Identifier (UUID), which will identify a certain TA amongst many that may be available in a TEE. Instead of having one main entry point like normal applications, given usually by the main function, a TA has to implement a set of entry point functions, collectively called the TA Interface, that can be invoked by the CA, to update the trusted application about life cycle changes.

The first function that will be called is the `TA_CreateEntryPoint`, which is only called one time during the whole life cycle of the TA, precisely when the TA starts for the first time. The function, that serves as a clean-up function, is the `TA_DestroyEntryPoint`, and is also invoked one time only, just before closing the TA. When the CA opens or closes a session for a specific TA, the `TA_OpenSessionEntryPoint` and `TA_CloseSessionEntryPoint` are invoked, respectively. The `TA_InvokeCommandEntryPoint` of a TA is invoked for each command invocation from a CA. The TA receives all the necessary arguments from the CA that will be used by the command function, within the TA, based on the identifier selected by the CA. The data received is assigned to that specific session, through the use of the data types defined by the API specifications, like the `TEE_Param`.

Using these functions and data type specified by the API, the source code has a guaranteed compatibility with other TEEs that are compliant with the GlobalPlatform TEE specifications.

2.3 Resource Management

This section provides some details from both the Linux System as well as from Genode. Since this work uses Linux as the normal world OS and Genode as the secure world OS, it is relevant to explore details of both systems. Therefore, we will focus on the memory aspect in this chapter, to allow a clear understanding of how both systems manage their resources.

2.3.1 Linux

The memory management has been one of the most important parts of operating systems, for quite a long time, since there has always been the need for more memory than the one physically available. The most successful solution implemented is the virtual memory. In the Linux, both user and kernel memory are independent and isolated. This isolation is due to the way the address spaces are used by the system, i.e., instead of using the hardware physical memory, the address spaces are virtualized, becoming abstracted from the physical memory available. This allows many virtual address spaces to exist, each assigned to a certain process. The kernel itself exists in these virtual address spaces, using a portion of it for certain tasks like storing data or determining whether a certain address is a kernel or user-space address. The virtual mapping of memory addresses also allows the existence of many

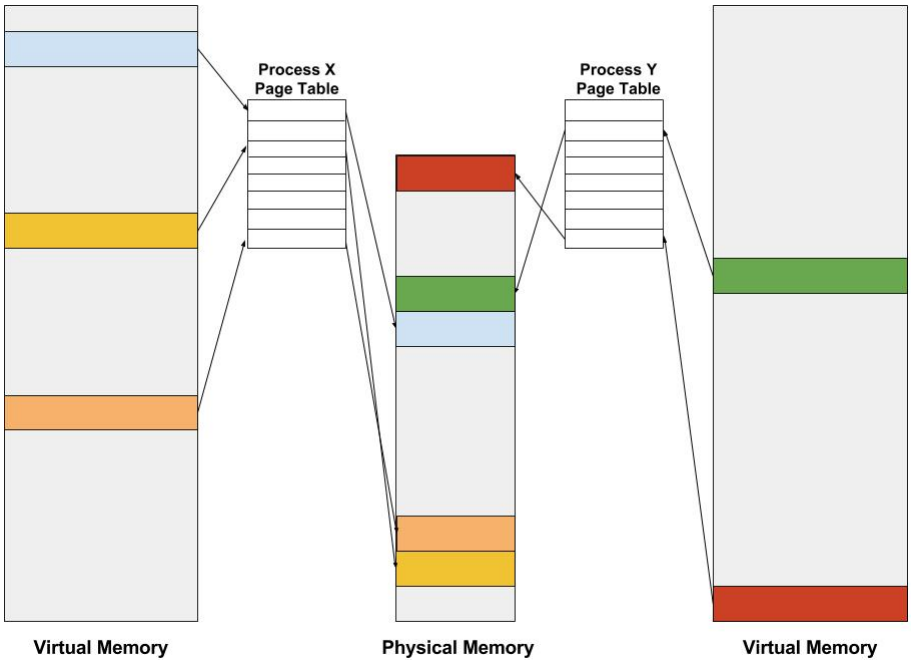

Figure 2.6: Abstract model of virtual to physical address mapping.

processes, each with an independent address space, linked to a relatively smaller physical address space.

To map virtual memory into physical memory, the underlying hardware uses page tables to find the correct match, as seen in Figure 2.6.

Being able to specify the memory for a certain process can easily overwhelm the underlying physical memory, since a large portion may be constantly being used aside from possible new memory demands that may come during the execution of the operating system. To avoid the inefficient use of physical memory the operating system must take actions, like saving the physical memory by only loading virtual pages that are currently being used by the executing program. This process is called Paging, but in Linux it is normally called Swapping. It consists of moving a page from physical memory, that has not been used frequently or for quite some time, to a special file called the swap file. This special file is located in a slower storage device, like a hard disk. This action results in leaving the most used or frequently used pages available in physical memory.

Access to the swap file is very slow, compared to the speed of both processor and physical memory. To avoid performance issues, like waiting to read a desired page from disk, the operating system must work with the need to write pages to disk to be used later on with the need to retain them in memory to be possibly used in a small slice of time. The goal is to have an efficient swap scheme that would make sure that all processes have their working set in physical memory.

Linux uses a Least Recently Used (LRU) page aging algorithm to choose pages which might be removed from the system. Every page contains information regarding its age, which is changed with every access to that page. The more that a page is accessed, the younger it is. Consequently, the less the page is accessed, the older and better candidate for swapping. Of course, not all pages are possible candidates for swapping. Pages containing kernel code responsible for interrupt handling or page table

management, should never be swapped and are, therefore, permanently in memory.

2.3.2 Genode

In Genode [5], the traditional approach to resource management is not the same as in Linux. Instead of employing the traditional strategies to abstract the physical resources and uphold the illusion of unlimited memory, by swapping memory pages to disk, the Genode OS simply arbitrates the access to such resources and allows the delegation of authority over resources between components. The low-level physical resource are represented as dedicated services, provided by the core component, which is the first user-level component, directly created by the kernel. As such, it represents the root of the component tree, having access to the raw physical resources such as memory, CPUs, interrupts, among others. In Genode the physical memory is represented by the RAM service of the core component. The concept behind this service is the same as a bank account, where a RAM session is comparable to a bank account. Each RAM session starts with the amount stipulated by the RAM session's quota and decreases with each ”purchased” item, i.e., the action of allocating physical memory from a RAM session can be compared to purchases in a store, where we exchange a certain amount of money located in the bank account for an item. Each piece of allocated physical memory is called a dataspace, where each dataspace is a container of physical memory that can be used to store arbitrary data.

Genode also presents its own resource manager. At boot time, the core component creates an initial RAM session with a balance containing all the available physical memory for the init component, the first and only child of core. After its creation, the init component can request authority over its RAM session and the core component can then delegate that capability. The init component may then create a new RAM session for each child component created after it receives full control over his own RAM session and transfer a certain amount of RAM from his own quota to its children. Each child can then request to gain control over his own RAM session, from the init component. This is done through a parent interface available, thereby gaining full authority over the memory budget that was assigned to it. A child component does not have the authority over init's RAM session nor the RAM sessions of any siblings. Each child may continuously subdivide their budgets in order to spawn grandchildren using the same procedure as their parent did before. The opposite procedure occurs in case a certain RAM Session is closed. The core component destroys all dataspaces allocated from the closed RAM session and transfers the RAM session's budget to the original RAM session, from where it was allocated, since the parent's reference account for all his children is maintained after the creation of a child.

2.4 Summary

In this chapter we explored the background information for this work, in order to provide a necessary insight on the details to come, further ahead. In the next chapter we will explore the related work of this work, delivering an overview on the current state of ARM TrustZone projects available at the moment, as well as development frameworks that use its capabilities.

3 Related Work

This chapter provides an overview on the current state of the art in TrustZone systems, in terms of concepts and ideas, followed by an analysis over some development frameworks for ARM TrustZone. Ultimately these sections will lead to an overall discussion, based on the details explored throughout this chapter.

3.1 State of the Art in TrustZone-based Systems

In this subsection, we provide details on the research and investigation of solutions being developed for TrustZone, how they are designed, and what sort of added value they offer. As we have seen in the previous section, ARM TrustZone technology has a partitioning mechanism, with two different execution environments. These are the main focus on the current research, namely how novel mechanisms may take advantage of one of these environments or even both, exploring the features provided by ARM TrustZone, whether its execution is in normal or secure world.

Several of these solutions focus on assuring services with more security measures, like authentication mechanisms, one-time passwords, use of cryptology, while others focus on exploring virtualization features. Last, some solutions focus on simplifying their design, reducing their trusted computing base (TCB), in turn reducing the attack surface they face. The majority of the solutions presented in this section are grouped as trusted applications. The boot sequence for all these TrustZone-enable system begins in the secure world, as referred previously, initiating the necessary modules, such as monitor, TEE OS, Trusted Applications, among others. The sequence is then followed by the normal world boot sequence, where the REE modules are initiated, namely the REE OS, user applications and every normal world application that needs proper clearance to access the secure world through the TrustZone driver. After the system finishes booting and is running, the execution world can be switched by executing the *smc* instruction, as described above. Figure 3.1 illustrates this entire process. They explore the TEE to communicate with the REE, to gain access to the necessary peripherals, either through the TrustZone driver or directly through the interrupts described earlier.

3.1.1 Security Services

DroidVault [17], a trusted data vault allows the secure management of storage and usage of sensitive data in untrusted Android devices, by creating a secure channel between remote data-hosting servers or data owners, and end users, allowing the owners to control access to sensitive data through the enforcement of security policies, while maintaining a small TCB. The sensitive data, while idle in the untrusted Android device, is encrypted until used in a secure data processing environment by trusted

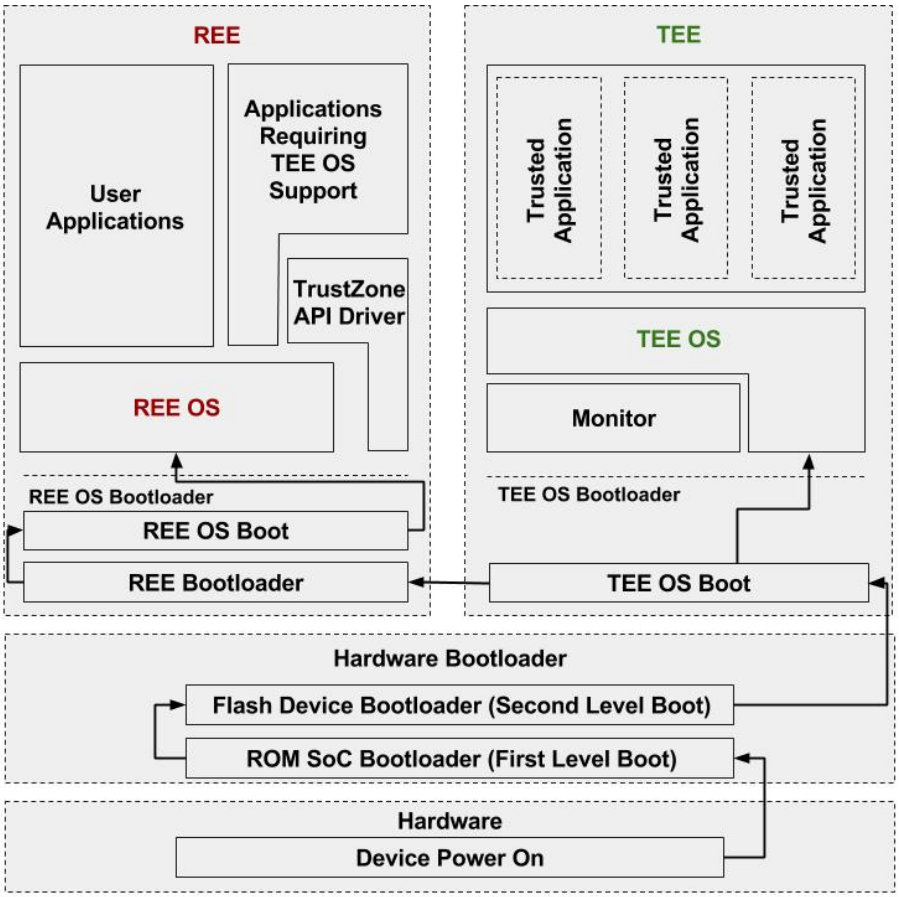

Figure 3.1: REE and TEE software and hardware overview.

code, which is then decrypted after its transmission to the TEE OS. This secure environment is one of four services that DroidVault supports and that enables data owners to extend their trust from their workplace, like a cloud storate service, to the untrusted Android device. Through the use of ARM Trustzone, and its TEE, DroidVault behaves as a small trusted engine that handles sensitive data operations between the TEE OS and Android OS located in the same mobile device. Balancing between full-fledged functionality and having a small TCB, roughly 12K LOC, DroidVault combines ARM TrustZone primitives with the untrusted Android stack. This data vault has three components: the Data Processing Module (DPM), I/O module and bridge module. The DPM is responsible for data transmission and data operations, maintaining a secure channel with the remote storage server to transmit sensitive data, through the use of asymmetric cryptology and authentication mechanisms. The data is cyphered before leaving DroidVault and its secure environment, but before any client applications can access this data, they must first be validated, the DPM verifies whether the loaded code is signed by the data owner, which then allows the code to execute certain operations, if given authorization, using the sensitive data. The bridge module exposes interfaces that facilitate communication between both worlds, using certain permitted functions and serializable data, through the use of shared memory. This module also allows DroidVault to use resources from Android, like the network and file system. The I/O module enables secure input and display. When requested by the DPM, the I/O module may display sensitive data directly to users, using a serial console, and receive user inputs. In terms of performance, the main overhead is due to

the context switch, as well as the data encryption, although the context switch depends on the hardware being used, which has impact on the SMC interrupts, context saving and restoring.

TrustOTP [18], a one-time password solution that focus on offering secure one-time passwords (OTP) tokens for smartphones that achieve both the flexibility of software tokens and the security of hardware tokens, through the use of ARM TrustZone. The basic concept behind OTPs is the automatic generation of numeric or alphanumeric string of characters that authenticates the user for a single transaction or session to an authentication server, being this way resistant against replay attacks. This framework is installed in the secure world and is composed by three components: OTP generator, secure display controller and secure touchscreen driver. The first is responsible for handling the secure generation of one-time passwords, given one OTP algorithm, like the time-based OTP (TOTP) or the event-based OTP (HOTP). The second component, secure display controller, is responsible for continuously displaying the generated OTP, by copying the image from a secure and reserved framebuffer to the display device, even in the event of rich OS compromise, since the rich OS cannot access this framebuffer, but must instead use its own. Since the peripherals are shared by both worlds, normal and secure, the current state must be saved and the display cleaned by TrustOTP before passing from one domain to the other, which will then restore the previous state of the current domain, i.e., the state of the rich OS must be saved and cleaned before TrustOTP can show the OTP. After showing it, its state must be saved and the screen cleaned before returning the rich OS state. Last but not least, the third component, the secure touchscreen driver serves the purpose of a secure input interface for the user to insert the password information in the mobile device, thus, the user can directly input its sensitive data in the secure domain without having to go through the untrusted rich OS. During the secure boot sequence, TrustOTP is loaded in the secure world, where it remains in its secure memory, only then does the secure boot load the non-secure bootloader, granting this way, isolation between worlds.

TrustUI [19], a novel mobile device trust path design offers secure interaction between end users and services based on ARM TrustZone. By applying techniques like cooperative randomization of the trusted path and secure delegation of network interaction. By running a small and secure kernel in secure world, TrustUI achieves a small TCB, and since it excludes device drivers for input, display and network from the secure world, but instead reuses the existing drivers from the normal world, it further reduces the TCB in size and allows it to adapt to many devices. The goal of TrustUI is protecting interaction between cloud services and end users, without significant resources. In its design, TrustUI splits, logically, the device drivers into two parts, the backend part, which is the unmodified driver, running in the normal world and the frontend part, running in the secure world, that interacts with the backend, allowing secure apps to safely access the device. This interaction is mediated by two proxy modules running in both worlds, that exchange data using a shared memory. The security of this data exchange point is guaranteed by the TrustZone monitor, but other potential attacks, regarding privacy and integrity rely on some of the features offered by TrustZone, like the secure boot, as well as those offered by TrustUI, like the LED and display color randomization, to prevent framebuffer overlay attacks, or keyboard randomization, to prevent information leakage or tampering. In terms of network secure, TrustUI has a SSL library located in the secure world and delegates all network-related function calls to the normal world proxy, to be called in the untrusted normal world OS, guaranteeing no information leakage, but since TrustZone does not possess a secure storage, it is up to the vendors and developers to choose which peripheral can be accessed only in the secure world, in order to achieve a trusted path from a mobile device application to a cloud service. Although the trust can only be fully assured by a bug-free SSL encryption

and decryption, since any anomaly might present vulnerabilities that may be exploited to break security barriers.

Trust-RKP [20], is a security system that aims at mitigating some Hypervisor problems. By running solely in secure world, its security monitor, which is the entire TCB of this security mechanism, is protected against attacks that may compromise the kernel, running in the normal world, consequently allowing it to prevent full control over system functions in the normal world, by forcing functions to go through the secure world to gain access to sensitive system functions and execute. Otherwise, it could increase the risk of kernel compromise, since the secure world does not fully control targets running in normal world, which could allow an attacker to modify kernel code and, possibly, bypass secure world verifications, as well as execute user space code in privilege mode. Although these features are solid, they rely on several assumptions, which may not be granted at some point, like for example the whole system loading in a secure way. Without verifications, this whole system might become vulnerable in case an attacker discovers a vulnerability, in which he can tamper with the secure boot, consequently, bypassing the initial integrity checks.

3.1.2 Virtualization

vTZ [21], a trustZone security extension system aims at enhancing isolation between virtualized TrustZone instances, by enabling TEE multiplexing through the emulation of one trusted virtual machine (TVM) for each virtual machine (VM) running in the normal world. Since TrustZone was not initially designed to be virtualized, but instead to be used by a single machine, regardless of the number of VM's running under it, this approach mitigates exposure of other VM's to possible attacks, like a guest VM accessing a secure memory region, triggering a page fault, which results in an unauthorized operation, being prohibited afterwards. The mitigation is due to every VM having their own virtual secure world, instead of having to share the real secure world, provided by ARM's processors, with all other VM's. This approach may bring increased security, but may also penalize the device overall performance since it will surely consume more resources, since there has to be one TVM for each VM running, which may bring scalability concerns and increase power consumption. Besides the solutions mentioned above, there are others that focus on providing similar security measures, but focusing on reducing the size and complexity of the TCB, either by applying simplified but assuring measures in the secure domain, or by deploying their solutions in the normal domain.

3.1.3 Design Simplification

TrustDump [22], a TrustZone-based reliable memory acquisition tool provides a secure mechanism to retrieve both the CPU registers, as well as the RAM memory of the rich OS, even in the event of a crash or compromise. This tool resides in the secure world, hereby ensuring a secure isolation from the normal world, provided by the TrustZone, so that the rich OS, possibly compromised, may not interfere in the memory acquisition process, allowing the TrustDumper module to successfully perform online malware analysis, as well as acquiring the memory of the rich OS, check the integrity of the OS kernel and transmit all the data retrieved to a remote server, for further offline analysis, since mobile devices have limited resources. This data can be sent either by a fast Micro-USB port, in case the normal world

OS is still operational or by a slow serial port, in the event of a normal world OS crash, although the context switch request may not be sent to the secure world in case of a normal world system failure, thus, the memory acquisition tool cannot be triggered in a reliable fashion. Using the Micro-USB instead of the serial port results in an increase in performance by 40 times. Given the simplicity of this mechanism, naturally, it's TCB is also small, with no need for a hypervisor or a kernel module in the normal world OS, since it only consists of a small memory acquisition module and basic malware analysis functions, allowing the use of a simpler secure world OS, instead of a full-featured one. Since this tool is labeled as a forensic evidence extraction tool, it serves the principal of leaving the extracted data untainted, since it does not required any code in the normal world OS to work as intended. The crucial point is to guarantee that TrustDump is securely activated in the boot process, in order to perform trustworthy memory dumps, thus it is assumed that the TrustZone code is secure. Also, since the micro-USB port that is used to transmit data lies in the untrusted REE, the data retrieved can be manipulated, therefore malicious code may target and compromise the memory dump, defeating the purpose of the tool's existence. Therefore, it must be verified, using an encrypted hash value, in order to guarantee that it was not tampered with. Besides this constraint, the Micro-USB driver may suffer from denial-of-service (DoS) attacks, in case the normal world OS is compromised, blocking its data transmission.

TrustICE [23], a TrustZone-based isolation framework, who's main objective is to provide isolated computing environments (ICEs), but rather than extending the secure world, this approach focuses on the normal world, relying on TrustZone to ensure secure code isolation from untrusted and possible malicious code running in the REE, through the use of ICEs. The main idea is to relieve the secure world of an increased risk of exposure by maintaining the TEE unchanged as well as the TCB to a minimum, by moving the secure code from the secure world to the normal world, entrusting all security measures to the trusted domain controller (TDC), located in the TEE. When the secure code is running on one ICE, both the REE as well as other ICEs are suspended and cannot access the resources in the active ICE, and when one ICE is suspended and thus, its memory is also in stand-by, the REE and any other secure code cannot, again, access the memory of the suspended secure code. This is done by the TDC, which is responsible for loading the secure code into an ICE, enforcing secure isolation between the secure code and the REE and achieving secure switching between an ICE and the normal world OS, which is required due to both being in the REE. In order to load the secure code, the TDC must first verify the integrity of both the secure code being loaded as well as the code from the ICE that sent the switch context request. Only then does the TDC load both codes into a secure memory region protected by a Watermark mechanism, offered by the NXP i.MX53 QSB, the SoC used for this work. As we see, there are quite a few trade-offs to consider. First, the number of assumptions made is quite crucial to the overall security of the system like handing over the secure code execution to normal world isolated environments, based on security checks provided by TrustZone and the TCB, disregarding the secure world isolation benefits. The assumption that the secure world could become increasingly exposed, due to having more applications running in the secure world, although valid, may be effortless, since the TEE is always a desirable target, regardless of the number of applications installed, since it may contain sensitive data, even though the ICEs are suspended upon the context switch. Another point is the way the ICEs execute, namely the isolation created, based on the suspension of both the REE as well as other ICEs, which may decrease the overall performance of the system, possibly increasing both response times, as well as power consumption. Besides, if an attacker succeeds at breaking the normal world OS, he can directly attack any ICE, tamper with it, and use it as a weapon to deceive the user into

providing sensitive data, like passwords or banking credentials. Since both the ICE and the normal OS are running in the REE, they share the same privileges, which may allow an attacker to tamper with the ICE memory, tamper with the secure code image, located in the device persistent storage or even delay the switch to secure world, through DoS attacks.

TrustICE was developed by H.Sun et al., the same authors of [22] and [18], which may indicate that this framework was used in the development of these solutions.

3.1.4 Other Solutions

In addition to the solutions mentioned above, focused on providing more security services, offering support to virtualization services or focusing on simplifying their solution, reducing their TCB, there are others, in which we will explore in a brief manner. DAA-TZ [24] is a DAA[1] scheme solution for mobile devices capable of using TrustZone, allowing remote service providers to authenticate a mobile user's trusted status or legitimate information without disclosing a user's identity. AdAttester [25] is a reliably ad fraud detection and prevention system. By providing attestable proofs to ad providers, bot-driven frauds and interaction frauds can be detected. The ad providers can define a group of policies to determine whether a sequence of ad requests are compliant with the predefined policies and whether or not they should be paid. By using ARM TrustZone, this solution avoids the need to trust the mobile software stack, relegating such constraints to the TrustZone technology. SeCReT [26] is a framework that builds a secure channel between the REE and TEE, reinforcing the access control to resources in the TrustZone. By providing a technique that enables TrustZone to protect a certain memory area from the compromised kernel in the REE, this solution makes use of existing hardware component without adopting additional software stacks, like a hypervisor. Trusted Language Runtime [27, 28], or TLR, is a system that protects the confidentiality and integrity of .NET mobile applications from OS security breaches, using the ARM TrustZone hardware features, to provide an isolated trusted environment. It provides runtime support for the secure component based on a .NET implementation for embedded devices, offering productivity benefits of modern high-level languages, such as strong typing and garbage collection, to application developers. SPROBES [29] is an introspection mechanism for ARM TrustZone-enabled hardware, capable of providing a view, not forgeable, of the normal world's processor state at any given moment. This mechanism enables the secure world to dynamically break into any normal world kernel routine, trap it and assign a trusted handler, in the secure world, to mediate that routine, being also able to restrict normal world kernel execution to approved kernel code pages, even if under the control of a rootkit. With these capabilities, it facilitates the monitoring of the normal world, since it allows the secure world to select the normal world instructions for which it has to be notified, receive the processor state at that moment, and finally, perform any monitoring actions desired before returning to the normal world.

There are also more uses to ARM TrustZone technology, although we only cover a few, due to the vast options available. C.Marforio et al. [30] devised a location-based second-factor authentication solution for modern smartphones, in the context of point of sale transactions, for the detection of fraudulent

[1] Direct Anonymous Attestation is a cryptographic protocol that enables a trusted platform to remotely authenticate itself, whilst preserving the user's privacy.

transactions caused by card theft or counterfeiting. F.Brasser et al. [31] present a systematic approach for restricted space hosts, in order to remotely analyze and regulate guest device usage in the restricted space, where policies applied are decided by the hosts that control the restricted space, being enforced by a trusted mechanism that executes on the smart guest device, using ARM TrustZone. Javier Gonzalez [32] conceived a set of run-time security primitives that enable a number of trusted services in the context of Linux, mediating any action involving sensitive data or sensitive assets in order to guarantee their integrity and confidentiality. C.Dall et al. [33] present KVM/ARM, an implementation of the Linux ARM hypervisor. This solution allows unmodified guest operating systems to run on ARM multicore hardware, through a split-mode virtualization feature that allows a hypervisor to integrate with the Linux kernel and split its execution across varying CPU modes, consequently allowing the leverage of existent hardware support and functionalities for Linux, in order to simplify the development and maintenance of hypervisors, while using current ARM hardware technology to run virtual machines with comparable performance results to native execution.

3.2 Development Frameworks for ARM TrustZone

As referred in the previous subsection, the evolution of software development for mobile devices led to the exploration of ARM TrustZone. N.Asokan et al.[34], explain several reasons that lead to the lack of potential exploration of TEE in mobile devices, some of which concern privacy and lack of trust issues between OEMs[2] and their willingness in letting third parties into their security environments. Other concerns focus on the lack of standardized APIs and available SDKs, but the open-source software reputation increase brought the open-source platforms, centered in exploring TrustZone capabilities. As seen in the previous Chapter, GlobalPlaftorm has its own TEE standard and has been trying to improve them ever since, as more and more projects apply their standards. With their efforts on TEE specifications[3] this issue may be diminished in the future.

This subsection will explore, some already created, platforms for the exploration of TrustZone features. Each framework will be analyzed in terms of architecture, compliance with GlobalPlatform standards, whether they allow hardware heterogeneity, some problems and some features.

3.2.1 Open-TEE

Open-TEE [35] is designed to work as a daemon process in the user space. This platform is compliant with GlobalPlatform specifications and can be divided into two components, the Manager and the Launcher, both of these processes descend from the Base process, which contains all the functionality of the TEE. The Manager is responsible for the communications between Trusted Applications (TA), monitoring their state, providing secure storage and controlling shared memory regions, basically functioning an operating system. The Launcher serves the purpose of creating new processes for upcoming TAs. After its creation, the Launcher loads a shared library implementing the TEE Core API, as stated

[2] Original equipment manufacturer (OEM), company that makes a certain component that is then marketed by another company typically as a component of this company's product.

[3] TEE System Architecture. http://www.globalplatform.org/specificationsdevice.asp

by GlobalPlatform, which will be used by the new TA processes, cloned from the Launcher, upon receiving a signal from the Manager. The library preload done by the Launcher improves the new process creation performance, since all common components are already loaded, leaving only a minimal effort in configuring and starting the new process, reducing the necessary time to do so. After its creation, a new TA is reparented to the Manager, so that it may control the TA and enforce GlobalPlatform policies, like the release of resources, in case the process crashes.

This platform achieves some heterogeneity, since it is compliant with GlobalPlatform standards and does not emulate TEE hardware, making it independent of TEE hardware. They were able to test one TA developed in Open-TEE, by compiling and running it in Trustonic's <t-base environment [36], another platform that is also compliant with GlobalPlatform standards for TEEs. As for TEE hardware, it allows the developer to configure the usage of resources, based on the hardware being used, by exploring the GlobalPlatform internal core API. In some cases, the target may be running a different TEE from the one offered by Open-TEE, with different hardware specifications for a certain purpose, like cryptographic accelerators with different CPUs, at different clock speed and throughput characteristics, requiring additional performance tuning. Although the authors do not explicitly state whether or not this framework uses ARM Trustzone, they were able to use it in ARM processors, and given its hardware independency it may be able to explore Trustzone without great effort.

3.2.2 OP-TEE

OP-TEE [37] is an open source TEE, developed by STMicroelectronics[4] and Linaro[5]. Like Open-TEE, this framework is compliant with GlobalPlatform TEE System Architecture specifications, also providing the TEE Internal core API v1.1 as defined by the GlobalPlatform TEE Standard for the development of Trusted Applications. It consists of three components: the normal world user space client APIs (optee_client), a Linux kernel TEE device driver (optee_linuxdriver) and the Trusted OS (optee_os). The available source code, with the corresponding documentation regarding build settings can be found at their public repository[6]. One of the major tasks was the abstraction of platform-specific parts that should allow OP-TEE to be ported and incorporated in products from different vendors in a fairly easy way. Currently, this platform supports several different SoCs, some are not publicly available, but others are, namely ARM Juno Board, NXP i.MX 6 Ultra Lite EVK, ARM Foundation FVP, Hikey Board, Texas Instruments DRA7xx and QEMU. Details are provided in their repository[7].

3.2.3 SierraTEE

SierraTEE [38] is a secure operating system developed by Sierraware[8] for ARM TrustZone hardware security extensions. Also like OP-TEE, this platform is both compliant with GlobalPlatform Internal and Client APIs and development tools, as well as open-source, being also able to support the full suite

[4] STMicroelectronics. http://www.st.com/web/en/home.html.
[5] Linaro. http://www.linaro.org.
[6] OP-TEE. https://github.com/OP-TEE.
[7] OP-TEE Platforms supported. https://github.com/OP-TEE/optee_os#3-platforms-supported.
[8] Sierraware. https://www.sierraware.com

of ARMv8 extensions for multicore and power management. As for normal world OSs, SierraTEE can integrate with Android, Linux, BSD and other ”normal world” operating systems using the hypervisor [39] feature introduced, where this hypervisor allows three distinct operation modes: paravirtualization for ARM TrustZone-enabled devices, paravirtualization for ARM11 and Cortex-A9 devices and hardware virtualization for Cortex-A15, allowing equipment manufacturers to choose the appropriate mode for their processor architecture and virtualization requirements. The first mode allows high-level OSs to run concurrently, on top the hypervisor, with minimal modifications, having minimal impact in performance. The second mode, offers multi-core management, both symmetric and asymmetric multiprocessing with a small number of non-intrusive hypercalls in the guest OS, reducing overhead. With the paravirtualization technology, the modifications required for existing applications are minimal. The third mode allows the emulation of hardware without changing the guest OS, with lower system impact than paravirtualization. The secure OS provided supports several SoCs, including mobile platforms, set-top boxes and automotive applications, clearly stepping up as a cross platform approach. It also supports mixed mode operation, allowing for legacy 32-bit ARMv7 OSs to co-exist with ARM64 TEE enabling equipment manufacturers to quickly transition from 32-bit to 64-bit systems. The OS also a variety of applications such as mobile payment, device authentication, and DRM for high definition 4K content protection to be properly secure, using the TEE capabilities.

3.2.4 T6: Secure OS and TEE

T6 [40] is an open source operating system, like SierraTEE, developed by TrustKernel[9] for TEEs in mobile devices using ARM TrustZone. As rich OS, T6 supports Android, Linux and other UNIX based operating systems, to run simultaneously and provides a strong security property for the legacy operating systems, aiming at providing an easy-to-use trusted computing platform for the research community and a product-quality TEE for mobile device providers. This OS has a small code base, roughly 6K LOC, giving it a small TCB, with a small attack surface, where the TrustZone general API is available, allowing the developer to take advantage of all its features. Although the authors do not offer concrete details and documentation on how it works, they ensure that it is possible to achieve other features like fault isolation, multitasking, secure user interaction through secure display, user input and network connection. The virtualization and secure boot features are still in progress, at the moment, leaving an attack window that may cause severe damage to the entire system, although other features like GlobalPlatform's TEE API is provided, ensuring compatibility with other standardized TEEs. This OS supports a few development boards, namely Samsung Exynos4, Samsung Exynos5, NXP i.MX 63, Versatile Express and ARM Fast Models.

3.2.5 ANDIX OS

ANDIX OS [41, 42] is an open source, multitasking, non-preemptive and ARM TrustZone aware operating system, that runs in the secure world of the processor and provides a TEE based on the GlobalPlatform TEE standards, like the previous frameworks. As rich OS, ANDIX supports both Linux

[9] TrustKernel. http://trustkernel.org.

and Android running next to it, also being able to assign peripherals solely to the secure world, thus isolating these peripherals from the normal world. Architecturally, ANDIX focus on minimizing the components responsible for creating the TEE, thus minimizing the TCB, like many of the solutions described earlier, to reduce the number of security flaws, consequently, reducing the attack surface. The ANDIX OS consists of many components, like the ANDIX kernel, the ANDIX secure user space components, and the ANDIX normal world support components, in order to provide a stable and secure TrustZone aware operating system. The kernel, located in the secure world, was designed as a small, monolithic kernel that provides process isolation, scheduling and communication facilities. It runs in kernel mode, providing a secure execution environment for TAs running in the secure world in the user mode. Taking advantage of the ARM Trustzone security extensions, the kernel is also capable of securing certain interrupts, therefore taking control over certain peripherals, as well as providing communication mechanisms between normal world applications and TAs, using SMCs.

Through the use of GlobalPlatform standard APIs for trusted applications and clients, ANDIX aims at providing source code compatibility between trusted applications written for ANDIX and other TEEs, compliant with GlobalPlatform standards. Besides these, ANDIX also provides a collection of user space runtime libraries component which is the basis for implementing the TEE runtime library. These include secure libc syscalls, newlib[10] and tropicSSL, needed to provide cryptographic functions.

ANDIX currently supports the i.MX53 Quick Start Board (i.MX53 QSB) as primary hardware platform, being also able to use it with a modified version of the QEMU processor emulator, using extensions based on the Qemu TrustZone implementation [43], with very poor performance results.

3.2.6 Genode

The Genode OS Framework [5] is a tool kit created by Genode Labs, with the purpose of allowing the creation of operating systems with extended security measures. The architecture in which Genode was build resembles ARM platforms, which allows Genode OS to take full advantage of CPUs with TrustZone technology, at a level that allows Genode to be used as a TrustZone monitor, through a hypervisor, that leverages the protection mechanism of ARM TrustZone. This allows program execution to execute in sandbox environments, with access control execution equivalent to normal world-secure world environments, i.e., granted only the access rights and resources that it requires to fulfill its purpose. In Genode, each component has a budget of physical resources assigned by its parent, unlike traditional operating systems, which create an abstraction from physical resources. This resource management allows the use of dedicated resources by the components, that are within the designated limits, or to assign parts of their resource quota to its children. The usage and assignment of budgets is managed by each component rather than a global policy of the OS kernel. Each component can also communicate with other components and trade resources if need be, but only according to specific rules, reducing the attack surface of security-critical functions, compared to other operating systems.

Genode supports existing software, like APIs, runtime environments and virtualization. This way, it's possible to deploy Genode on a variety of already existing kernels from the L4 [11] family, like NOVA,

[10] newlib. https://sourceware.org/newlib/

[11] L4 - microkernel family used to implement Unix-like operating systems, as well as other systems.

Fiasco OC, Codezero and others, along with the possibility to use it on top of the Linux kernel for development-test cycles. The framework also comes with a custom microkernel for ARM-based platforms, like Genode, reducing the complexity of its trusted computing base.

Since Genode is a collection of small building blocks, it also provides ready-to-use components like device drivers for SoCs as well as other components. This allows developers to choose a small existing block and start to build their module or system, on top of that component. To aid early developers, new to the framework, Genode provides some small tutorials, as well as some documentation on the matter at hand, although it may not suffice sometimes, since small details, unknown to developers unfamiliar with Genode, may cause several problems. To mitigate these difficulties, Genode has a mailing list for the community to discuss problems that arise from any developer using Genode and, although very helpful, sometimes it is still not enough, since we may reach the limitations of the framework itself, at that point. On those occasions, the developer is on its own, which may be a major setback in the developer's aspirations. During the development of this work, such problems arose, since most of the documentation provided is directed for a specific hardware, while the framework supposedly supports a few different SoCs.

3.3 Discussion

In terms of the current state of the art in TrustZone systems, researchers and developers aim at providing small security modules that have increased functionality, allowing their solution's TCB to decrease, both in size and complexity. Therefore, their attack surface is reduced, becoming less attack prone, sometimes to the same attack surface level of the secure world itself, for solutions installed on the TEE and in most cases the entire solutions turns out to be less complex, which might bring performance gains. Other types of projects, try to explore other capabilities of ARM TrustZone like its support for virtualization, or its secure boot to ensure safety measures over their solution. Finally, for solutions installed on the REE, their attack surface varies, based on their assumptions and security mechanisms interaction with the normal world, as well as their TCB complexity and size. All solutions make certain assumptions, in order to successfully apply their solution.

DroidVault assumes that certain attacks are beyond their scope such as a malicious user subverting DroidVault's integrity by using hardware attacks, like Direct Memory Access attack or peripherals, cold-boot attacks or by compromising the hardware integrity. They also assume that the ROM code is secure and cannot be tampered with (secure boot). TrustOTP also assumes that the ROM code is secure and cannot be tampered with, but also that some limitations, due to restrictions imposed by mobile device manufacturers, can have impact in some security aspects. TrustUI assumes that mobile devices hardware is secure as well as all software running in secure world is safe. Cannot prevent physical attacks like reading data from memory. Trust-RKP assumes that the system is loaded securely, the kernel functions properly using mutual exclusion memory mapping and the hardware platform implements the privileged eXcecute Never (PXN) memory access permission, according to ARM specifications. vTZ assumes that ARM TrustZone technology software is secure. TrustDump assumes that the ROM code is secure and cannot be tampered with. TrustICE assumes that the secure code may have bugs, which may allow the attacker to compromise one ICE and then target other ICEs, and that an attacker cannot

Solution	Domain	Focus	Authors Tested In	Assumptions
DroidVault	TEE	Security Services	NXP i.MX53 QSB	Secure ROM (Boot)
TrustOTP	TEE	Security Services	NXP i.MX53 QSB	Secure ROM (Boot)
TrustUI	Both	Security Services	Samsung Exynos 4412	Mobile device HW & TrustZone SW are secure
Trust-RKP	TEE	Security Services	Samsung Galaxy S5 & Note 3	Secure ROM (Boot), Mutual exclusion memory mapping & PXN memory access permission
vTZ	REE	Virtualization	Unknown	TrustZone SW is secure
TrustDump	TEE	Reduce TCB	NXP i.MX53 QSB	Secure ROM (Boot)
TrustICE	REE	Reduce TCB	NXP i.MX53 QSB	Secure code may have bugs & Physical attacks not possible

Table 3.1: TrustZone state of the art solutions overview.

access physical mobile devices or launch physical attacks, like removing the Micro SD card. A summary can be seen in Table 3.1. Finally, several of these TrustZone-based solutions enrich the users and developers that follow, to the extent that they allow them to explore these mechanisms through the use and development of trusted applications that run on the TEE.

In terms of development frameworks, Table 3.2 contains an overview of the frameworks addressed earlier, on how they compare in certain aspects. The standardized column refers to compliance with GlobalPlatform's Internal and Client APIs, as described throughout this section. The TZ column refers to the use of ARM TrustZone capabilities by the frameworks. The documentation column refers to available public information regarding each framework, on their usage and functioning method. In this aspect, low designates almost no reliable documentation available, medium means some documentation is provided although the quality is insufficient and high the sufficient documentation provided with good content. The support column refers to both quantity and quality of the community support given, whether in general forums or mailing lists. In this aspect, low means either designated forums and mailing lists do not exist or do not provide much help at all, medium means the given framework has some community support in the type of mailing list or forum, but the help provided is insufficient and high means that the designated framework has good community support, with high quality help. Finally, the available column, refers to the availability of the framework's source code. In this aspect, yes means that the source code can be easily acquired, no means that, at this moment it is not possible to download the source code and agreement refers that, in order to download the source code, it is necessary to fill a form and accept a license agreement.

As we can see, there is not an ideal development framework, that suits every software developer's needs. Although almost every framework follows GlobalPlatform standards, they also present some trade-offs between them, namely the amount and quality of the documentation available, as well as the

Framework	Standardized	TZ	Documentation	Support	Available
Open-TEE	Yes	Unknown	Medium	Medium	Yes
OP-TEE	Yes	Yes	High	Low	Yes
SierraTEE	Yes	Yes	Medium	Medium	Agreement
T6	Yes	Yes	High	Low	No
Andix OS	Yes	Yes	Medium	Low	Yes
Genode	No	Yes	Medium	Medium	Yes

Table 3.2: Development frameworks overview.

available community support, for example, OP-TEE provides a good amount of documentation, with very detailed information on the steps one should take, but in terms of community support does not offer much since there does not exist a dedicated forum, where one could place its own doubts and problems, as opposed to SierraTEE, for example, that has its own forum, where developers exchange knowledge with each other.

During this work, we tested two of these frameworks, namely Genode, since it is used frequently for TrustZone projects and Andix, which looked promising, since it was compliant with GlobalPlatform API standards. The first framework presented several difficulties in its usage, mostly due to compatibility issues, or lack of good documentation. As for the second framework, since it was stated in the author's thesis that this framework was used for both the NXP i.MX53 QSB and QEMU, we were hoping to achieve the same results using our NXP i.MX53 QSB, but all the available source code was directed only to QEMU, and so, mostly due to the lack of documentation and available source code, we did not progress much with this framework. Eventually, we returned to Genode, which was then used until the end of this work.

3.4 Summary

In this chapter we explored the current state of the art in TrustZone systems, in terms of ideas and concepts, followed by existing development frameworks for ARM TrustZone and the capabilities they offer, as well as documentation and support quality. In the next chapter we introduce the architecture details of our system.

4 Architecture

In this chapter we will go through the capabilities and design of TrustFrame. We will explore in detail the functionality of the system and its components. We will start by tracing a threat model, where we describe possible attacks to the system, followed by a detailed explanation of the system's overall architecture and composition, where we explain how the components are connected, how do they share data and what it allows developers to do.

As stated previously, the main goal of this system is to offer a SDK, that is easy to use for both development and testing of applications, capable of using ARM TrustZone without the need for additional effort from the developers that use the system.

In terms of requirements, this system should meet the following requirements:

1. **Ease the development process** by allowing developers to immediately start developing their solutions, saving time and effort in the process.
2. **Compatible with the GlobalPlatform API specifications** by offering new possibilities for developers that are already familiarized with the specifications and for those starting on this journey.
3. **Work on real hardware**, allowing more and more developers to use and test the developed solutions by themselves, assuming they use the same hardware, required by the system.
4. **Allow efficient execution of applications on top of it**, allowing developers to build their desired solutions, without having to sacrifice some of its capabilities, due to performance limits.

4.1 Threat Model

There are some scenarios where security issues may raise alarms. First of all, in case an attacker compromises the normal world OS, i.e., Linux, he is able to execute arbitrary calls through the client API, where at some level may lead to a DoS attack, ultimately crashing the system. An attacker may also feed arbitrary input data, in an effort to assess response times and system behavior, breaking the Non-interference property[1]. As for attacks to access sensitive data or to compromise the secure world OS, we sought to mitigate as much as possible by verifying everything that is sent to the secure world. One vulnerability to take into consideration could be direct memory access (DMA) attacks, since the system uses the *dma-mapping* library to create a shared memory region for both worlds. Normally, in

[1] Non-interference: A program has the non-interference property if and only if any sequence of low inputs (low sensitivity) will produce the same low outputs, regardless of what the high level inputs (highly classified) are.

these types of attacks, the target is the physical addresses used by DMA. Traditional Hypervisors use the CPUs MMU to limit software access, which does not prevent alternative Bus Masters[2] from getting the memory, and so, by circumventing the CPU protection, DMA attacks succeed. In this system we use ARM TrustZone, which mitigates these attacks, since ARM TrustZone extensions always tag transactions with the NS bit. Therefore, the protection is assured not only by the ARM CPU, but also by its extensions, one of which is the Advanced Microcontroller Bus Architecture, as referred previously, precisely avoiding alternate Bus Masters to gain access to protected memory regions. As for other vulnerabilities, we trust and assume the following components are secure:

1. **ARM TrustZone software and hardware.**
2. **The NXP i.MX53 QSB**
3. **Genode and U-boot's source code.**
4. **The developed source code.**

4.2 System Components

Throughout this document, we already discussed some of the major components of this system, namely its two OSes, Linux in the normal world and Genode OS in the secure world and the hardware in which the system runs, the NXP i.MX53 QSB. We now head into further detail around these components, as well as other details not yet fully explored, but also present as seen in Figure 4.1.

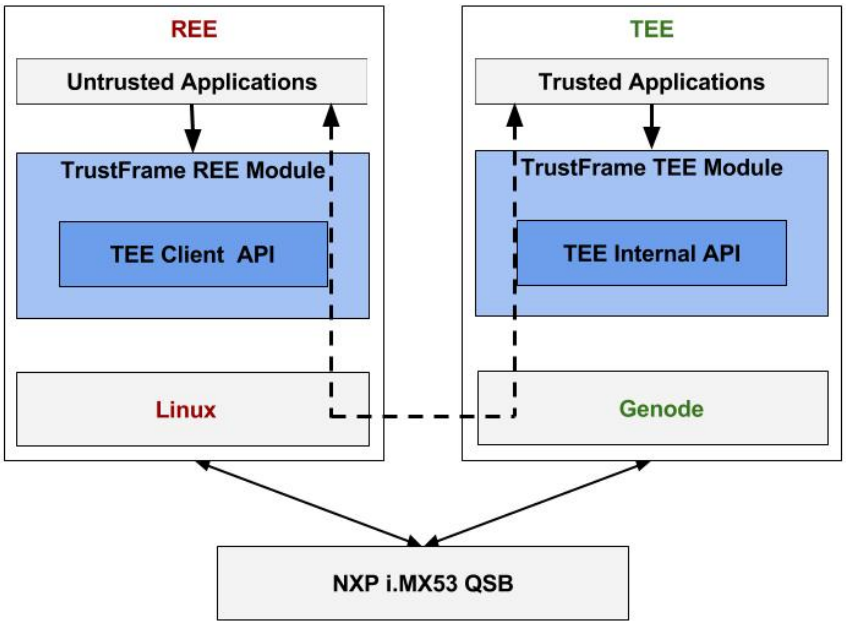

Figure 4.1: TrustFrame architecture overview.

[2]Bus Master. Program, either in a microprocessor or in a separate I/O controller that enables a device connected to the bus to initiate input/output transactions.

In terms of execution flow, the system starts with the boot of the SoC used in this work, the NXP i.MX53 QSB. By default, the board boots in the secure world, where it will load Genode's micro kernel, containing 20KLOC, which compared to Linux's 2016 expectancy of around 20MLOC, is quite an achievement. This micro kernel was specifically developed for Genode, thereby reducing the complexity of the trusted computing base compared to other kernels. As seen in the previous chapter, Genode is a collection of small building blocks, where new add-ons can be seen as services to the kernel core. One of these services already provided is the *tz_vmm*, which is Genode's TrustZone demo. In this demo, a VM session interface is used to allow a user-level virtual machine monitor (VMM) to affect the whole state of the CPU of a non-secure virtual-machine (VM), initiate a world switch to the non-secure world, and obtain the VM's state, after an exception-triggered return. To take advantage of Genode's existing work with TrustZone, the *tz_vmm* demo was used as a foundation in this system. After Genode's init process finishes, the *tz_vmm* service immediately prepares to boot the normal world and initiates a world switch,, where it will load a custom Linux kernel. Since Genode merges the Linux binary image with its own generated image, in our setup, we changed Genode to use a custom Linux image, compiled by us, instead of the original precompiled image, therefore static. This allows features to be added to the kernel, and allowed us to add our system call which is responsible for the data transfer between both worlds. Another interesting fact is that the Linux root filesystem is built using an initial RAM disk[3], which allowed us to add our binaries, that would be used in the normal world, like a client application. When Linux finishes booting, it is ready to be used, like any Linux distribution, and at this point, the developer can them call his program.

In the normal world, a client application invokes the TEE Client API, implemented by us, which in turn calls our system call, in order to transfer the client application's data. This system call is responsible for executing security verifications, creating the shared memory region, executing the *smc* instruction and finally, returning any received data back to the client application. In the secure world, each time a client application executes and a system call initiates a world switch, the *tz_vmm* receives the normal world data to be handled. It then calls our GlobalPlatform API handler, responsible for preparing the received data and redirecting the client application request to the correct function call from the TEE Internal API, implemented by us, where that data will be manipulated and a response will be prepared to be sent back to the normal world.

4.2.1 Client Application

The client application (CA) is the normal world side of the application that will use ARM TrustZone in our system. This part focuses on calling the GlobalPlatform TEE Client API, using the specification's data types as arguments. Each data type has a specific role within the development process. Some of these types are basic data types, containing simple identifiers or values for a certain task, while others have a more complex role. There are also some data types which are implementation-defined, as referred previously. The CA must prepare the necessary data types, in order to pass these types as arguments. Since these data types are mostly structures with basic types within, the CA must initialize them correctly, in order to allow the TEE Client API to properly retrieve the data contained within.

[3] Initial RAM disk (initrd) is an initial root file system that is mounted prior to when the real root file system is available.

4.2.2 Trusted Application

The trusted application (TA) is the secure world side of the application that that will use ARM TrustZone in our system. This part focuses on calling the trusted services, specified by the client application. These trusted services must be previously exposed, to allow the client application to know what services it may call. The TA does not invoke the TEE Internal API, but is informed by it instead, each time the trusted OS calls TEE Internal API. This allows the TA to receive information about life-cycle changes and to relay communication between Clients and the TA.

4.2.3 Hello World Example

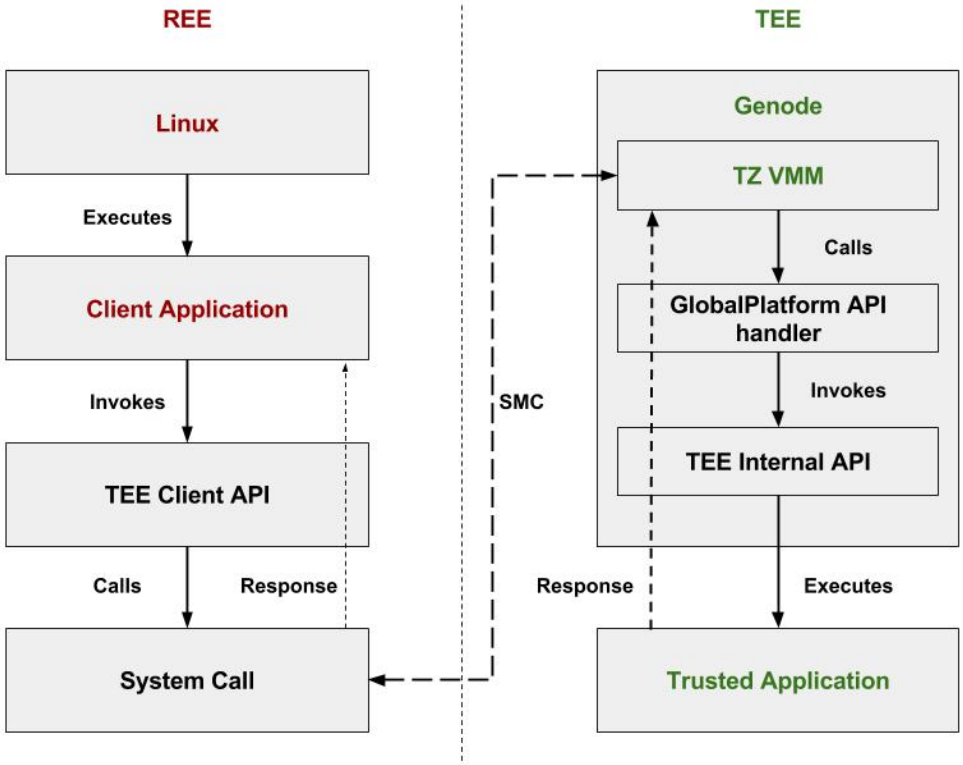

Figure 4.2: TrustFrame execution flow.

The system already has an example of a Hello World program, in which it calls the GlobalPlatform API, compatible with its specifications. This example is the combination of the client application that runs on Linux with the trusted application that runs in Genode. The entire execution flow of our example follows the execution flow represented in Figure 4.2. In this example, the program connects to the secure world by invoking the API functions, which in turn call our custom-made Linux system call that creates the shared memory region with the secure world and stores the data to be manipulated, by a specific command function implemented in the secure world. This command function is selected in the client application, by passing a certain identifier, previously assigned. In our example there are three possibilities to be chosen from, and we execute each command one time, to exemplify different ways to

pass different parameter types. After the secure world finishes the received task, it stores the response in the same shared memory region and switches back to the normal world, where the result is displayed.

4.3 Development Process

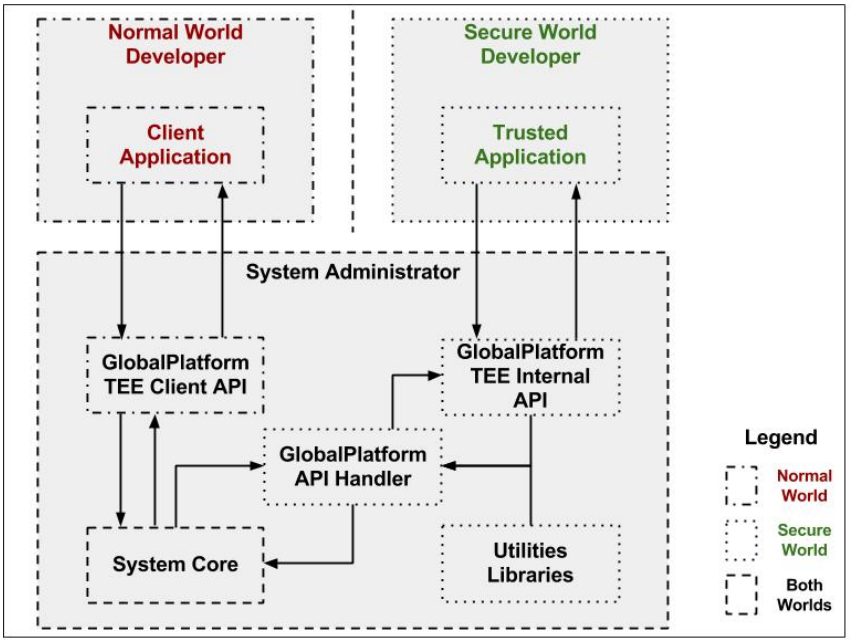

Figure 4.3: Developer responsibilities in the system.

In our system, we sought to ease the life of developers, and therefore, we structured our system in a way that allows three types of developers to act, each with a specific role:

1. **Normal world developer**: responsible for the implementation of the client application, using the GlobalPlatform TEE Client API invocations specified in our implementation of the standards.
2. **Secure world developer**: responsible for the implementation of the trusted application, with the services to be called upon by the client application, using the utilities offered by our system.
3. **System administrator**: responsible for extending the implementations of both the GlobalPlatform TEEC API and the utilities libraries in our system, as well as the GlobalPlatform TEE Internal API, contained into the main building block of our GlobalPlatform API handler, where all the logic is implemented.

These roles and responsibilities are represented in Figure 4.3. With this structure, any secure world developer does not have to mandatorily reimplement the TEE Internal API. He is only required to implement the services he wants to offer, so that a client application may call upon them. Along with the services, he must also define the identifications (IDs) of those services and UUID of the trusted application. He must then deliver those IDs to the normal world developer, so that the normal world developer may know what to call. The system administrator works like a bridge between both world

developers. If necessary, he may add features and extend the existing system, in order to adapt the system to new requirements. It is also possible that these three roles may be performed by a single developer, which was the case during this work. Although all the necessary logic for the presented system is already implemented, in case the system suffers extreme changes in its core, or require additional features to support new requirements, it may be required to adapt and change the TZ VMM service, in order to support any new behavior.

4.3.1 GlobalPlatform APIs Implementation

As already referred previously, during this work, not all the functions specified in the GlobalPlatform API were implemented. Since the API specifications are quite extensive, with several details to take into account, we chose to focus on the primary and most important functions.

The following functions from the GlobalPlatform TEE Client API were implemented:

- `TEEC InitializeContext(const char* name, TEEC Context* context)`: This method initializes a new TEE context for the communication, between client application and the TEE specified. The TEEC API defines that multiple TEEs may be supported and it is up to the developer to map a certain name to a certain TEE. A context is a logical container, that links a client application to one of these TEEs.
- `TEEC FinalizeContext(TEEC Context* context)`: This method finalizes a TEE context and cleans all resources associated with it. It is only called when all Sessions inside this TEE context have been closed.
- `TEEC OpenSession (TEEC Context* context, TEEC Session* session, const TEEC UUID* destination, uint32_t connectionMethod, const void* connectionData,TEEC Operation* operation, uint32_t* returnOrigin)`: This method opens a new Session between the Client Application and the specified trusted application. The target trusted application is identified by a destination parameter, containing the trusted application's UUID. The function's connection parameters allow the implementation of access control methods for a given client application session, but such methods were not implemented in the system. The returnOrigin parameter describes the origin from where the call returned. In case of success the session with the trusted application was successfully opened.
- `TEEC CloseSession(TEEC Session* session)`: This method closes a TEE session and cleans all resources associated with it.
- `TEEC InvokeCommand( TEEC Session* session, uint32_t commandID, TEEC Operation* operation, uint32_t*returnOrigin)`: This method invokes a command specified by the parameter commandID, within the given session. The command identifiers must be exposed by the trusted application. The operation parameter is a data type, specified by the API, containing the type and value of the parameters to be sent to the trusted application. The parameter values can either be integer values or shared memory blocks. These shared memory block may be temporary memory blocks or registered memory blocks. The register and allocation of these

memory blocks were not implemented, given that our system call already provides the necessary capabilities.

As for the GlobalPlatform TEE Internal API, the following functions were implemented:

- `TEE_Result TA_EXPORT TA_CreateEntryPoint(void)`: This function is executed once on start up when it creates a new instance of the trusted application.
- `void TA_EXPORT TA_DestroyEntryPoint(void)`: This function is executed once when the trusted application instance is destroyed.
- `TEE_Result TA_EXPORT TA_OpenSessionEntryPoint(uint32_t paramTypes,TEE_Param params[4], void** sessionContext)`: function is executed when a client requests to open a session with the trusted application. The arguments *paramTypes* and *params* may contain parameters specified by the client application, to be passed to the trusted application instance. These arguments can also be used by the trusted application instance to transfer response data back to the client.
- `void TA_EXPORT TA_CloseSessionEntryPoint(void* sessionContext)`: This function is executed when a client session is closed.
- `TEE_Result TA_EXPORT TA_InvokeCommandEntryPoint(void* sessionContext, uint32_t commandID, uint32_t paramTypes, TEE_Param params[4])`: This function is used to invoke a command issued by the client, within a given session. The trusted application can access the parameters sent by the client through the paramTypes and params arguments. These arguments can also be used by the trusted application instance to transfer response data back to the client.

Again, we refer that due to the extensive details and specifications provided by GlobalPlatform for both TEE Client API and TEE Internal API, we implemented just the functions described above. This opens new possibilities for future work and possibly new extensions, with new functionalities.

4.4 System Details

This section provides some details of our system. First we will provide information on the bootstrap process of our entire system (4.4.1). We will then provide the memory layout for this work (4.4.2), followed by an explanation on the parameter passing details concept of our system (4.4.3). Finally, we will explore some minor details of our GlobalPlatform module in the secure world (4.4.4).

This system is designed to ease the workload of its users, namely developers. By handling the use of ARM TrustZone technology by itself, this aspect is omitted, to the developer that uses the system, although present.

By analyzing the example provided, any developer that wants to use the system will have a basis, in which he can build his own solution. This way, the developer saves time and effort in his development process, since there is no need to worry on how to implement his solution to use ARM TrustZone, or how to exchange data between the normal world and the secure world, or even how to switch between

those worlds. He is only required to strictly focus on implementing the concept behind his solution, while maintaining compliance with the GlobalPlatform standards. In case the developer merely focuses on developing services in the secure world, his task does not require such strict compliance, since he is not required to implement the GlobalPlatform TEE Internal API again.

4.4.1 Bootstrap Process

During the initial bootstrap phase of the NXP i.MX53 QSB, our bootloader, *U-Boot*, loads the Genode kernel binary along with additional boot modules into the physical memory. These boot modules are chunks of data, like the ELF images of Genode's components: core, init, all components created by init, and the configuration of the init component. After this task finishes, *U-Boot* transfers control to the kernel, where the kernel then passes information about the physical resources and the initial system state to the core component. The core component then begins to execute its purpose, by making low-level physical resources (physical memory, processing time, device resources, initial boot modules, and protection mechanisms) of the SoC available to other components in the form of services. Then, it creates the init component, using its own services, and delegates all physical resources, and control over the execution of all subsequent component nodes, which can be further instances of init, as we already explored in chapter 2.

Since boot modules are read-only memory, they are called ROM modules, and can be used by passing the name of that module as a session argument during the creation of a session. As we already seen, in Genode, a component that obtains a session to the core component RAM service is a RAM client, and so, each client can make the ROM module visible within its local address space, by using this method. Usually, bootloaders on ARM platforms merely support loading a single system image and *U-Boot* is no different, although some efforts to overcome this limitation exist. Since a Genode-based system consists of potentially many boot modules, the system follows the concept of a monolithic image, i.e., merges all the used boot modules into a single image, that will then be loaded by *U-Boot*. In our system, to reflect our modifications on the Linux kernel, initial RAM disk along with Genode itself, we provide each component to Genode, who then merges everything into a single *uImage* binary, an image file that contains a U-Boot wrapper, that includes all the OS and loading related information. This *uImage* is then ready to be loaded into a MicroSD card and booted by *U-Boot* in the NXP i.MX53 QSB.

4.4.2 Memory Layout

In terms of memory layout, the development board used in this work contains two memory banks of 512MB each, making a total of 1GB. During the development of this work, it was noted that only one of those memory banks was in fact mapped by Genode, namely bank 0. Although we do not know why only one of the memory banks was mapped, we suppose that it may have been due to the lack of need, given by the simplicity and low memory requirements needed by the Linux kernel used, as well as Genode itself. The used memory bank goes from address 0x70000000 to 0x90000000, and is divided in half for each world, i.e., the memory addresses from 0x70000000 to 0x7FFFFFFF are mapped as secure and the memory addresses from 0x80000000 to 0x8FFFFFFF are mapped as non-secure. This

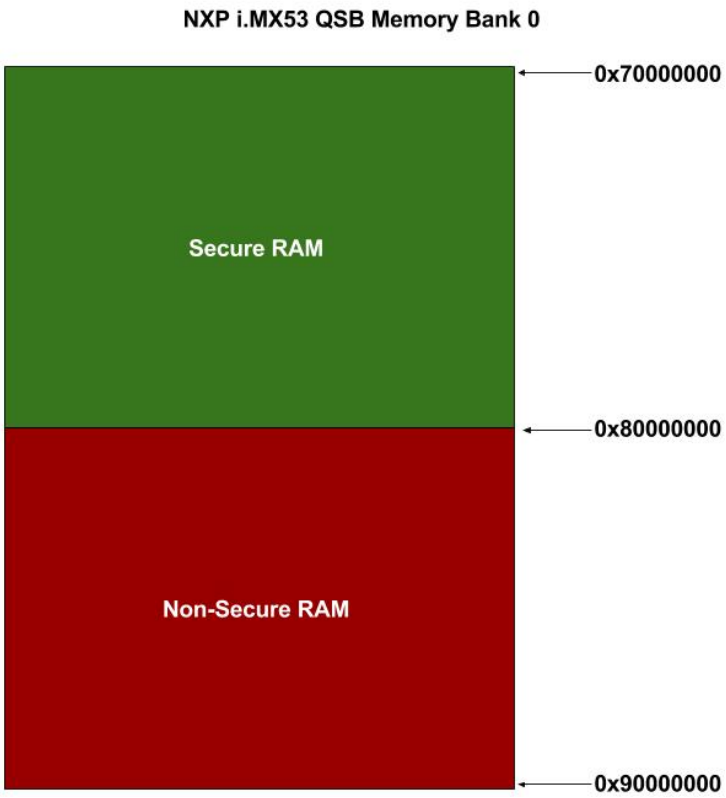

Figure 4.4: Memory mapping for the i.MX53 QSB bank 0.

memory mapping results in the assignment of 256MB for the secure world and 256MB for the normal world. Figure 4.4 illustrates the memory layout described above.

Component	Execution World	RAM Quota
init	Secure	235MB
tz_vmm	Secure	10MB
globalPlatform_server	Secure	5MB
Linux	Normal	96MB (77MB usable)

Table 4.1: RAM quotas for secure and normal world components.

As for the assignment of RAM quotas, each component in the secure world can be assigned a specific amount of RAM for its execution. Table 4.1 displays the RAM quotas in our system. The init and *tz_vmm* components, already explored previously, contain RAM quotas of 235MB and 10MB, respectively. The 235MB of init are transferred at boot to it, while the 10MB of *tz_vmm* are previously assigned in its run script. A run scripts is a script mechanism used internally by Genode, where all the compile targets, services and RAM quotas are located and assigned, allowing them to be used to integrate and test system scenarios. Our GlobalPlatform module is declared in the run script of *tz_vmm* as the *globalPlatform_server* service, with an assigned RAM quota of 5MB. This value was determined by looking at the RAM quotas of other services used by the *tz_vmm* component, which contained similar values. Finally, the last component is the REE OS, Linux, which has 96MB of assigned to it, despite

having 256MB to use in normal world. From these 96MB, only 77MB are usable since the remaining 19MB are reserved at boot. Of these 77MB, around 64% is free at runtime, roughly 50MB, to be used by applications. Although we do not have a concrete explanation of why the normal world OS is assigned less than half of its available memory, we believe that it may be have been with intent, due to the simplicity of the Linux kernel, since some of its functionality was removed, to allow the kernel to run in the normal world and also due to it being an older version (2.6.35).

4.4.3 Parameter Passing

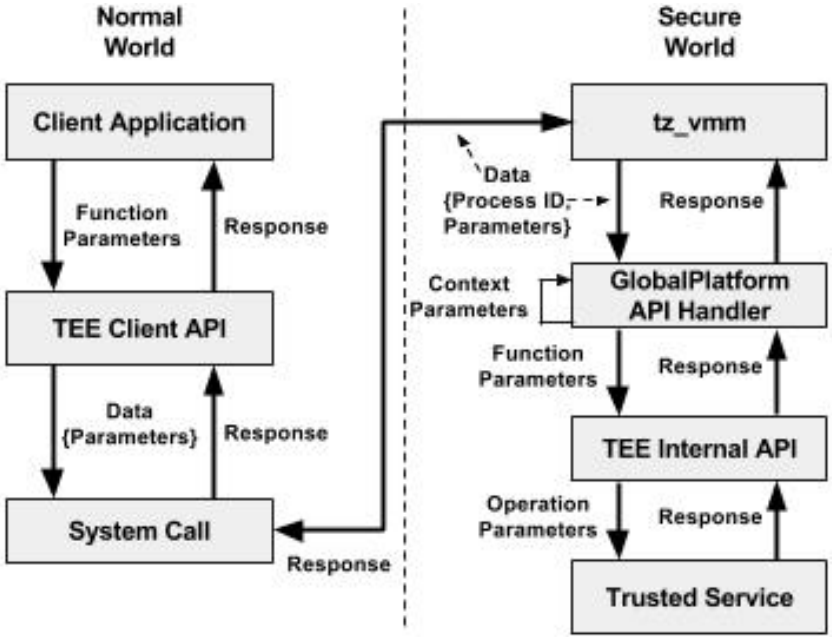

Figure 4.5: Data flow between both worlds.

We will now explore how parameters pass in our system, between both worlds. As stated previously, the GlobalPlatform standards specify certain data types, to be used during the invocation of the functions composing both world APIs. These data types contain the data that will be manipulated in the secure world, and require some preparation before being sent as an argument. This preparation is typically composed by the initialization of the data type with the desired values. More detailed information on this preparation is covered in the next chapter. Figure 4.5 represents the system's entire data flow scheme.

The parameter passing process starts in the normal world when a client application invokes a function from the TEE Client API, passing some data types as arguments. The TEE Client API function receives the parameters and prepares a container in which it will send these parameters. After the container is ready to be sent, the TEE Client function calls our custom system call and passes the data container to it. In turn, the system call prepares the received data container to be sent to the secure world, but first it adds the process identifier to it, so that it may be used in the secure world to differentiate sessions. The system call then stores this prepared data container in the shared memory region, so that the secure world might retrieve it. The system call then triggers an interrupt that will switch to the secure world, by executing the *smc* instruction. After this step, the system call awaits for a response, so that it

may return to the TEE Client API, who then passes the response to the client application. Further details on some of these important steps will later be explained in more detail.

In the secure world, the interrupt triggered by the *smc* instruction is redirected to the *tz_vmm* component who then retrieves the data container stored in the shared memory region. The data container is then passed to our GlobalPlatform handler, where it will be parsed, in order to retrieve the parameters. These parameters are not only the operation parameters, but also other parameters used to create a session context. The necessary parameters for that task are immediately used in this handler, before the handler calls upon the TEE Internal API. Only after this step finishes and the current session is able to proceed, will the handler call the desired TEE Internal API function, passing the operation parameters that it will use. In case the function called is the `TA_InvokeCommandEntryPoint`, the parameters will be handed to the specified trusted service, which composes the trusted application. This service is specified by the command function identifier, given by the client application.

4.4.4 GlobalPlatform Module

Our GlobalPlatform module contains five source files, where each source file has a distinct task. The main source file contains the implementation of our GlobalPlatform API handler, along with some necessary Genode source code. The Genode source code is required, in order to allow the *tz_vmm* component to use the remote procedure call (RPC) interface and call our handler. The TEE Internal API is implemented in another source file, to allow further changes to be executed without having to go through many lines of code. Therefore, it becomes easier for the developer to locate himself in the source file. The trusted application's services are located in another different source file. This allows the secure world developer to easily identify where he needs to add his trusted services, which will be called by the client application. Again, this allows the secure world developer to avoid exploring the entire source code of the module. The remaining source files contain utility functions which are used throughout this module, they are separated, specifically to provide better understating to developers, on where they are used, i.e., some are only used in the GlobalPlatform API handler, some are only used for the TEE Internal API and some are specific to the trusted services implementation. The developer, either the secure world developer or the system administrator, will need to explore these utilities, in order to take advantage of them and use them in their implementation.

4.5 Summary

In this chapter we explored the architectural details of our system, its components and how they are connected. We also explored an approach on different types of possible developers for this system, as well as the roles of each one. We finished the chapter by exploring some system details, concerning the transfer of data, as well as our GlobalPlatform module. In the next chapter we present the implementation details of this prototype, concerning the components and capabilities addressed in this chapter.

5 Implementation

This chapter provides in-depth details over the implementation of this work's solution. We will start by describing the path that we travelled during the development of this solution. We will also explore some choices on each of the major components of the system, already explored previously in a much broader approach.

5.1 Development Path

We started this journey by exploring some of the resources found in our related work, namely by testing both Genode and ANDIX. Since the existing experience with this line of work was none existent, we struggled to test some demo scenarios provided by each framework. We also felt some difficulties in replicating some tutorials found online, since most of these tutorials had a few years and used deprecated cross-compiling toolchains. Some of these toolchains were quite old and, therefore, difficult to find, while some newer versions did not work on the examples provided and some of them were extremely difficult to use, due to many incompatibilities, resulting in compile errors that should not exist.

With Genode, we started by trying the *tz_vmm* demo but without great success, which led us to ANDIX, where we saw potential to serve as a basis for this project, since ANDIX already uses both the GlobalPlatform TEE Client and Internal APIs, as specified in the standards. With ANDIX, the available documentation was focused on QEMU instead of the NXP i.MX53 QSB, although it was stated that it supported this SoC. After some time without progress, we opted to try to build our system by ourselves, using other means. We explored many community forums looking for answers, as well as documentation from the SoC manufacturer. We started by booting a Linux kernel in the secure world, without any modifications. After that step was successfully taken, we explored the world switch mechanism, which turned out to be quite more complicated than anticipated. Again, this step required extensive research, in order to understand if we were on the right path and executing all the necessary processor instructions required to successfully switch to the normal world, saving the processor's current state and loading the previous one in the process. Through this stage, we explored not only the documentation on the ARM processor, present in our board, but also on the board itself, since this particular board presented custom interrupt and trustzone controllers, which we will explore further ahead.

To boot the Linux kernel in the normal world we had to make some modifications, since the unmodified version would hang during boot-up, due to permission issues. This issues don't occur when the kernel boots in the secure world, since the entire kernel is in the secure world, resulting in full access rights and execution of sensitive instructions. In the normal world, the same principal cannot be applied,

since the idea behind using ARM TrustZone is to have access control over the normal world OS. Eventually we were able to successfully boot the same Linux kernel in the normal world, but we were having issues with the root filesystem, preventing us from having a stable system to start using as a foundation for our system.

5.2 Normal World Details

This sections provides implementation details concerning the normal world. We will focus on the normal world OS, Linux (5.2.1), as well as the system call implemented (5.2.2), which represents our logical channel, in which we communicate with the secure world. Afterwards, we will explore in more detail some implementation choices and concepts (5.2.3), along with some security details (5.2.4).

5.2.1 Normal World Linux OS

After a few unsuccessful attempts on establishing a stable system, we opted to apply the gained knowledge and tried to test the Genode *tz_vmm* demo again, which we successfully did this time. Genode already used a Linux kernel, fully modified to run in the normal world, alongside its micro kernel. As previously stated, if a normal Linux kernel booted without some modifications, the system would hang during boot. These modifications are required, in order to avoid access permissions issues that cause the system to hang. These issues would occur by executing sensitive instructions in some kernel component drivers like IPU, GPIO, as well as trying to access other peripherals, deemed critical. To allow the kernel to boot without these issues, the Genode staff had to modify these drivers, introducing hypercalls that occur whenever the Linux kernel tries to access a device, which is not assigned to the normal world. When this happens, an external data abort occurs, and the control is passed to the secure world, so that the device may be emulated. The basic idea of emulating device access is allow the hypervisor to pass control to the TrustZone virtual machine monitor (VMM) as soon as the non-secure OS accesses an address outside the allowed physical address boundaries. Afterwards, the VMM may then inspect the address and the program counter of the non-secure OS that raised the access violation. Based on the program counter value, the VMM can then fetch the faulting instruction, decode it and emulate it in software, bypassing the issue that did not allow the Linux kernel to properly boot.

Since these modifications were already done, instead of using our modified kernel which still lacked some modifications for this purpose, we opted to take that advantage and use it for our solution, allowing us to immediately continue with our work.

At this point we already had both world OSes ready, but lacked to understand if Genode offered a world switch mechanism to be used at will. Since we did not find any indications of an available mechanism to be used, unless we replicated the same assembly instructions the system already executed at boot, we opted to devise our own mechanism that could allow us to simply use it at will. We started by modifying Genode to use a Linux kernel image compiled by us, instead of using the same precompiled image that it normally used. This allowed us to modify the kernel that was used as the normal world OS, therefore, allowing us to implement our module which would allow the world switch.

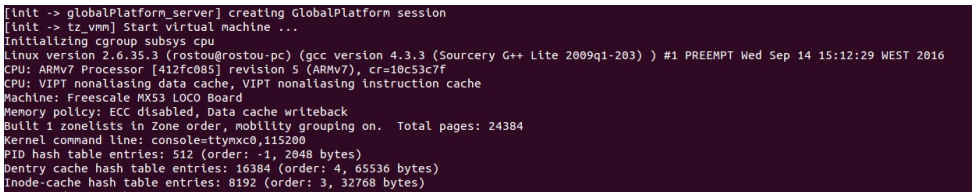

Figure 5.1: Linux starting to boot after Genode finished booting.

Figure 5.1 displays the moment Genode finishes booting and prepares to boot Linux, by creating a virtual machine where Linux will boot and run. After this virtual machine is created, by the *tz_vmm* component, the Linux image is loaded, and starts to boot.

5.2.2 Linux System Call

Since we could now add functionalities to our kernel and see them reflected immediately after booting both Genode and Linux, we created a custom system call. This system call initially served the single purpose of executing the assembly instructions necessary for the world switch. We took advantage of Genode's interrupt handling, which redirected world switches to *tz_vmm*, allowing us to change it and introduce our secure world handler, serving as the basis for the next steps of our implementation. Our system call allowed us to immediately switch from the normal world to the secure world when necessary. The major benefit was the simplicity introduced, since instead of having to replicate the same assembly instructions, over and over again (see Figure 5.2), we only needed to use a single line of code to call the system call. This system call would later be called by our GlobalPlatform TEE Client API implementation, each time a client application invoked a function from the API, as we will see further ahead.

```
asm volatile("mov r0, #4 \n"
             "dsb        \n"
             "dmb        \n"
             "smc #0     \n" ::: "r0");
```

Figure 5.2: Initial assembly instructions for the world switch.

After this important step, we needed to make some changes in Genode's *tz_vmm* demo, to redirect our system call's *smc* to our secure world handler. This handler would later serve the purpose of preparing the data received from the normal world to be manipulated in the secure world and the response from the secure world to be sent to the normal world, as we will see further ahead.

Afterwards, we wanted to pass data between both worlds, so that we could later send data from the normal world, manipulate it in the secure world and return it to the normal world. We devised a simple SQLite application, that ran in the secure world, as a service in Genode, where we would execute queries, given by the normal world. This SQLite application was the first step into understanding how we could later implement the GlobalPlatform TEE Internal API, since we needed to understand how Genode worked and how to communicate with other services.

Following the implementation of the SQLite application, we started to implement a mechanism that

```
asm volatile ("mov r0, #4 \n"
              "mov r1, %0  \n"
              "mov r2, %1  \n"
              "dsb         \n"
              "dmb         \n"
              "smc #0      \n"
              : /*no outputs */
              : "r" (genblk_buf_phys), "r" (size) /* inputs */
              : "r0", "r1", "r2"); /**/
```

Figure 5.3: Final assembly instructions for the world switch.

would allow us to pass some data from the normal world, through the system call, and into the secure world. After a thorough research, we successfully implemented this mechanism into our Linux system call. We started by creating a shared memory region, where both worlds could access it, using the *dma-mapping* library, which contains the direct memory access API. We already explored some security details in the previous chapter, concerning the use of DMA, so we will explore what it allowed us to do. The DMA is a method, provided by a DMA controller, that allows an I/O device to send and receive data directly to or from the main memory. This is done by bypassing the CPU, to speed up memory operations, since the CPU can process other tasks while the data transfer is being performed. This allowed us to obtain a certain memory region address, which we could pass as an input to the secure world, through the registers, allowing the secure world to receive that address and fetch the data. Thus, we achieved a way of passing data, although initially we only passed queries to be executed by the SQLite application. Later, after implementing our GlobalPlatform API handler in the secure world, we started passing parameters. We will explore this concept in more detail as we proceed in this chapter.

The changes we made to the assembly instructions, in order to pass the memory region address are reflected in Figure 5.3. In this figure, *genblk_buf_phys* is the address located in the DMA address space, which is allocated for the shared memory region. Both the memory region address and the size of the allocated region are stored in two separate registers, so that the secure world may know the boundaries, in which it may retrieve and store data. Another interesting detail is the first assembly instruction displayed in this figure, where we pass the number four as an argument. This particular number serves a purpose in the secure world, namely to allow Genode to identify this interrupt as our own. More details will be explained further ahead.

```
// SMC System Call
SMC_SystemCall(INVOKE_COMMAND, args);
```

Figure 5.4: Function call for the SMC system call.

To signal our custom system call we only need to use the single line of code displayed in Figure 5.4. In this figure we see a function that receives two arguments: the operation identifier and the arguments to be passed to the secure world. This function will then signal the system call and pass the client application arguments. Instead of immediately calling the system call, we opted to introduce this indirection to ease developers in understanding how to use the system call. This way, instead of having

to know the correct system call identifier, so that the correct system call is called, it is only required to call this function with the invoked API function identifier and its arguments. This allows the function to execute any required verifications, before actually calling the system call, reducing the workload of developers, that may want to extend the TEE Client API.

5.2.3 GlobalPlatform TEE Client API

Since we could now exchange data between both worlds, we started to implement the GlobalPlatform APIs for both worlds. We began by the normal world, where we researched how other frameworks implemented the API specifications according to their own concept. This research was done, since the client side of the API would surely be similar, due to its independence from the normal world OS and also to understand better how the specifications could be applied. We looked at the OP-TEE [37], since the framework's source code was publicly available. Since the API specifications already provided the necessary instructions for the header file, to be used by the TEE Client API source file, and since this header file was already available in the OP-TEE repository, we took advantage and used it, instead of having to create an exact copy by ourselves. Some of the header file specifications were left as implementation-defined on purpose by the GlobalPlatform API specifications. We maintained these definitions from the OP-TEE header file, since they would serve our purpose as well.

As for the API implementation, we used some minor ideas from OP-TEE as well, but immediately deviated to our own path, since our concept of using the Linux system call to pass data to the secure world diverged from OP-TEE's ideals. Since we needed to pass the data from the client application, through the system call, to be sent to the secure world, we opted to create a data container. This container contains all the arguments and parameters necessary for the secure world to manipulate them and return an accurate response. We followed the same principal for each function that comprises the TEE Client API. Each function executes the following tasks: receive the client application's arguments, store those arguments in the data container, call our system call and wait for a response. All the received arguments are stored into the data container, separated by certain special characters, that work as delimiters. These characters exist to allow the secure world to parse the entire container and, based on the character encountered, assign each piece of data to the correct destination. When the secure world receives the whole container, it will need to assign the correct parameters to the correct session and so on, which it does using this method, achieving an accurate assignment of the received data. After the API function finishes preparing the arguments, it calls our system call and transfers the prepared data container to the system call. Since the system call is located into the kernel space, it allows us to retrieve the process identifier that executed the program. The system call then appends that identifier to the received container and stores the data in the shared memory region. This step is always done to allow the secure world to differentiate between two processes that use the same parameters, like sessions identifiers and such values, that could normally overcome isolation barriers.

With our method implemented, we introduced the concept of instance into our system, as is also implied in the GlobalPlatform API. Using the same example above, but with our method implemented, we avoid compromising each request, by granting isolation between both requests. This way, different processes can now concurrently invoke commands using similar parameters, without compromising each other, since they will be redirected into different contexts, where each context will only handle the

parameters specific to both process and session identifier. More details concerning the manipulation of session identifiers will we explained further ahead, in Section 5.3.

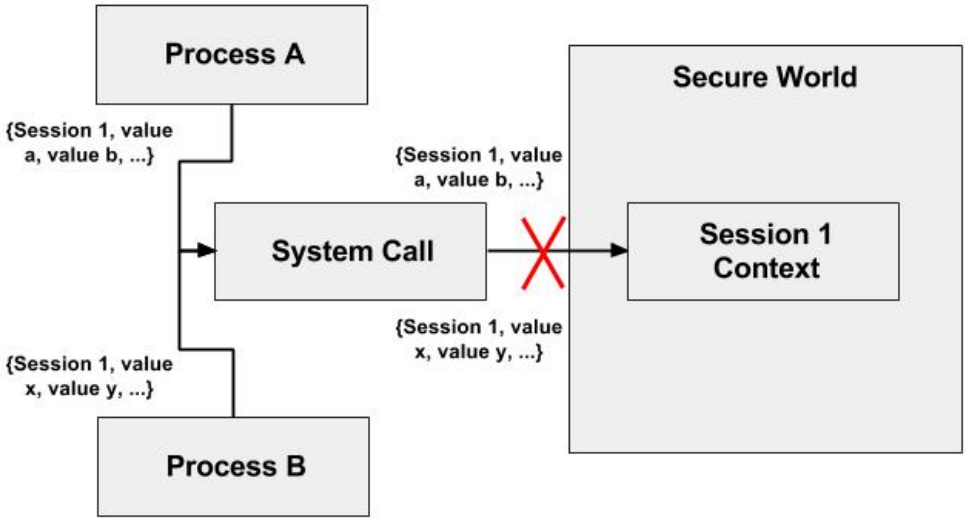

Figure 5.5: Wrong process isolation concept.

To exemplify what would happen if our concept was not implemented, we must consider an example where two processes would concurrently invoke the API functions, providing the same session identifier, but different parameters. This would probably cause problems for one or both requests, since they would access the same session context in the secure world and possibly overwrite themselves, providing inaccurate responses. Figure 5.5 illustrates this unwanted scenario. As opposed to the first, we should now consider an example where each request is processed isolated from each other, instead of possibly overwriting themselves.

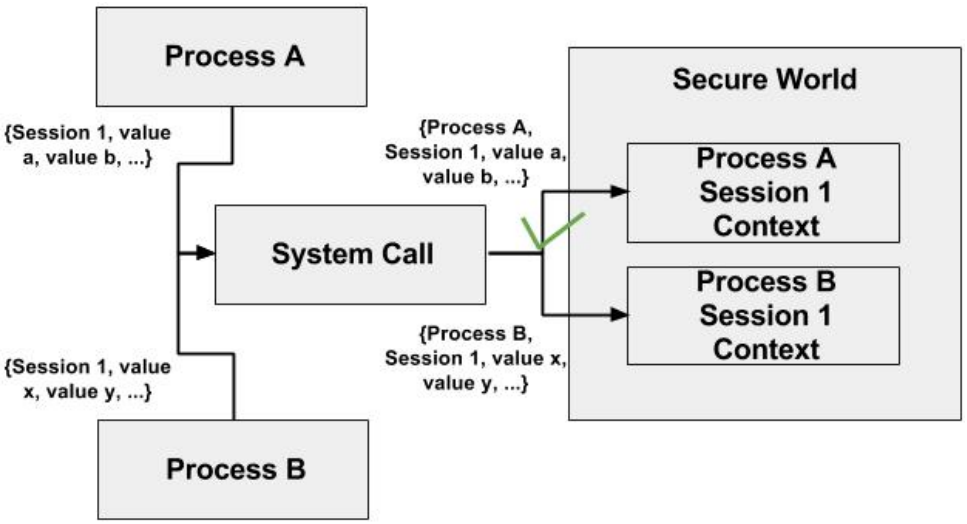

Figure 5.6: Implemented process isolation concept.

Figure 5.6 exemplifies what actually happens if two processes execute the API functions with the same session identifier in our system. In this case, having applied the concept of instance, we created

the necessary isolation barriers that allow both processes to only access their context, as is desired.

5.2.4 Normal World Security Details

Having explored the logical communication channel between both worlds, focusing on how they communicate and share data, we will now focus on some security details surrounding this important aspect of the system. We sought to enforce some security measures before calling the system call and before the system call creates the shared memory region and stores the received data in it, since these are two important phases of the communication channel.

Before calling the system call, we execute a simple input validation method, to avoid sending special characters that the system is not prepared to deal with. In case the arguments contain any on these characters, the input is ignored and a message is displayed, informing the user that the arguments contain strange characters. This is just a minor verification, to mitigate possible argument assignment issues.

When the system call is called, we move from user space to kernel space. Since user space data will be passed to the system call, verifications are required, to check if any variable coming from the user space is not trying to access memory regions outside the user space boundaries. To perform such validation and verifications, it is necessary to use a specific Linux kernel API specially designed for user space memory access. This API includes several functions, that allow such verifications, as well as the retrieval and sending of user space data. One of the most common used verifications is the use of the *access_ok* function. This is a special function which is used to check the validity of a pointer in user space. This function can verify if a user space application is asking the kernel to read from or write to kernel addresses. It is possible to specify a type of verification argument, which may be *VERIFY_READ* or *VERIFY_WRITE* for verifications on the readability or writability of the memory region, respectively. The *VERIFY_WRITE* type also identifies whether the memory region is readable as well as writable, implying the *VERIFY_READ* type, thus, it is not required to verify for readability if we already verify for writability. Such verifications were applied to our system call, to avoid undesired attacks on the kernel space.

5.3 Secure World Details

This section provides implementation details concerning the secure world. Since we already covered the most important details over Genode, we will focus on the modifications we made to the existing components, rather than Genode itself. We will start by our modifications to the *tz_vmm* component (5.3.1), followed by the GlobalPlatform API handler we created (5.3.2). Afterwards, we will explore the implementation details of the TEE Internal API (5.3.3). We will then provide some details on the secure world services that are developed by the secure world developer (5.3.4), followed by our Hello World application example details (5.3.5).

5.3.1 TZ_VMM Modifications

After the implementation of the normal world details, we started to prepare Genode for our requirements. We began by looking into the *tz_vmm* component, in order to understand how we could receive the normal world data container and use it in the secure world. We created a simple secure world handler that is called each time Genode's interrupt handler receives a specific argument, defined by us. This argument, as referred previously, is also sent in one register, which will allow identifying what to do when the argument is received. Since Genode already contains three defined behaviours for different types of interrupts, ours was the fourth, hence the reason the number four is sent in a register. This allows Genode to redirect the interrupt from our *smc* to our handler. This handler will then retrieve the arguments passed in the other registers, map the shared memory region using the received address with the size of the region and extract the data that was stored in it. From this point forward, our secure world handler contains the data sent from the normal world and is able to pass it down to our GlobalPlatform API handler, serving as a bridge between the normal world and our GlobalPlatform API handler.

5.3.2 GlobalPlatform API Handler

After we retrieve the data, we must then call our GlobalPlatform API handler. This handler was developed as a service to Genode, just like *tz_vmm* is, which allowed us to call it using Genode's RPC interface, adapted for our needs. Using this interface, we can pass the normal world data to the handler, which then reverse engineers the process executed by the TEE Client API. By detecting the special characters sent along with the data, the handler is able to partition the received data container and assign the data correctly. This allows it to identify which operation was executed in the normal world and, based on it, execute the respective TEE Internal API function, passing the necessary arguments. Besides passing the necessary arguments, this handler is responsible for handling data updates. These updates occur after a secure world service manipulates the data it receives, and so, this handler ensures an overall synchronization between all the secure world components that must deal with the data. In the end, the *tz_vmm* component retrieves the latest data updates and sends them back as a response, to the normal world.

This handler is also responsible for preparing the secure world session generation. This method consists in creating a random secure world session, based on both the process identifier and session identifier received from the normal world. This way, the secure world is able to differentiate two different requests for the same normal world session identifier, since they will match different secure world session identifiers. This is the implementation of the instance concept mentioned above, which allows the creation of isolation barriers.

5.3.3 GlobalPlatform TEE Internal API

The implementation of the TEE Internal API turned out to be simpler than anticipated. This was mainly due to the limitations of the API functions, since we could not change the structure of the function, like adding new arguments and such details. As a consequence of these limitations, much of the workload to be done in the secure world is already performed by our GlobalPlatform API Handler, thus

the API functions became simpler to implement. Since the TEE Internal API does not specify a concrete behaviour for each function, most of them serve the purpose of signalling that a certain operation occurred or is going to be executed. The exception is the *TA_InvokeCommandEntryPoint* function, which sets global identifiers, like which process identifier or session identifier executed the request and calls a custom function which redirects the function parameters to the desired service. This redirection was an implementation decision executed on purpose, to allow secure world developers to easily introduce their services in the system, without having to deal with any of the logic present in the GlobalPlatform API handler or even the TEE Internal API functions. This way, when a secure world developer wants to create new services to be called upon by the client application, he is only required to implement the logic of that specific service and add it to the available services pool. This can easily be done by following the same principal of the already available services, using less than five lines of code. He may even take advantage of a group of utilities already created, which do most of the necessary labour, like updating important data types, to be sent back to the normal world. More details on this matter will be explained further ahead.

5.3.4 Trusted Services

The implementation of trusted services is confined to a single source file, in our system. This organization derives from our ideal of separating the different possible roles of development and responsibilities, as referred previously. This way, any secure world developer can easily identify where he needs to add his trusted service. This single source file currently contains our trusted service examples, along with a special function, which redirects the TEE Internal API command invocation to the respective trusted service. A new developer may replace this source file with his own, provided that he adds this special function, containing the redirection to his trusted services. By looking at our example, he can easily replicate the redirection mechanism. This special function is required, since it is called by the *TA_InvokeCommandEntryPoint* function, from our TEE Internal API implementation. The secure world developer may even add his services to the existing ones, and add a redirection entry to the special function, without replacing the provided source file, further reducing his workload and, possibly, allowing him to easily test his solution, without having to worry about other details, like calling the special function correctly or know which Genode libraries he must use besides the ones needed for his services. Figure 5.7 contains a snippet from our request redirection function. In this example, we specify how the trusted application behaves if it receives a *TA_FUNC_CPY_STR* identifier. In our case, the client application will execute the service *copyStr*. This service copies the string located in the data buffer parameter, from the GlobalPlatform data types, which can then be used if needed, and copies a response string, which will be sent back to the client. This identifier must be previously configured, as we will see further ahead.

Besides the trusted services source file, the secure world developer will have to create a header file containing the command identification for his trusted services. This is required, in order to allow the client application to know which commands it may call upon. The developer must also generate a UUID, which will identify his trusted application among the existing ones. Although this UUID differentiation is not implemented, the creation of the trusted application UUID task belongs to the role of the secure world developer. An example of this configuration can be seen in the code sample displayed in Figure 5.8.

```
  // Request redirection function
  TEE_Result (*funcSelect)(uint32_t param_types, TEE_Param params[4]);
  case TA_FUNC_CPY_STR:
   PINF("Calling command function to copy a string\n");
   funcSelect = &copyStr;
   return funcSelect;
```

Figure 5.7: Example of a request redirection to a trusted service.

```
#define TA_HELLO_WORLD_UUID { 0x8aaaf200, 0x2450, 0x11e4, \
  { 0xab, 0xe2, 0x00, 0x02, 0xa5, 0xd5, 0xc5, 0x1b} }

/* The TAFs ID implemented in this TA */
#define TA_HELLO_WORLD_CMD_INC_VALUE 0
#define TA_FUNC_CPY_STR 1
#define TA_FUNC_PRINT_CMD_DATE 2
```

Figure 5.8: Example of a trusted application configuration.

After the secure world developer finishes implementing his trusted services along with the corresponding header file, he must provide this header file to the normal world developer, so that the normal world developer may call the trusted services, accordingly.

5.3.5 Hello World Example

After the implementation of both the GlobalPlatform TEE Client API and TEE Internal API, we needed to test them. Since we implemented quite a few modifications on the overall system, our SQLite example was too complex to start with, since it would need to be adapted again to our current needs. We chose to implement a Hello World example, that would allow us to exemplify that our system works as is expected. This Hello World example, comprises a CA, that uses the TEE Client API to perform certain tasks in the secure world and a TA, which contains trusted services which are called by the CA.

We initially based our example in some ideas researched previously at the time we were looking into OP-TEE, and continued from there. The initial example that served as a basis to our example only executed a specific type of commands in the secure world. It only passed values, to be incremented by a command function implemented in the secure world. We added more functionalities to the example, by allowing it to pass more types, like strings, instead of just numerical values. We then noted that possibly every client application would have a similar structure, since the GlobalPlatform API specifies certain return values, based on success or not, which are passed back from the secure world to the normal world. Therefore, every client application will have to verify the return type whenever it calls a function from the TEE Client API. These verifications are always the same, resulting in replicated code, which ends up creating a structure pattern. This is true as well when the client application will call the *TEEC_InvokeCommand* function to execute a command in the secure world. Every time this function is called, the application has to previously prepare the arguments to be sent. This preparation also follows a certain pattern, since the function arguments are specific types, specified by the TEE Client API,

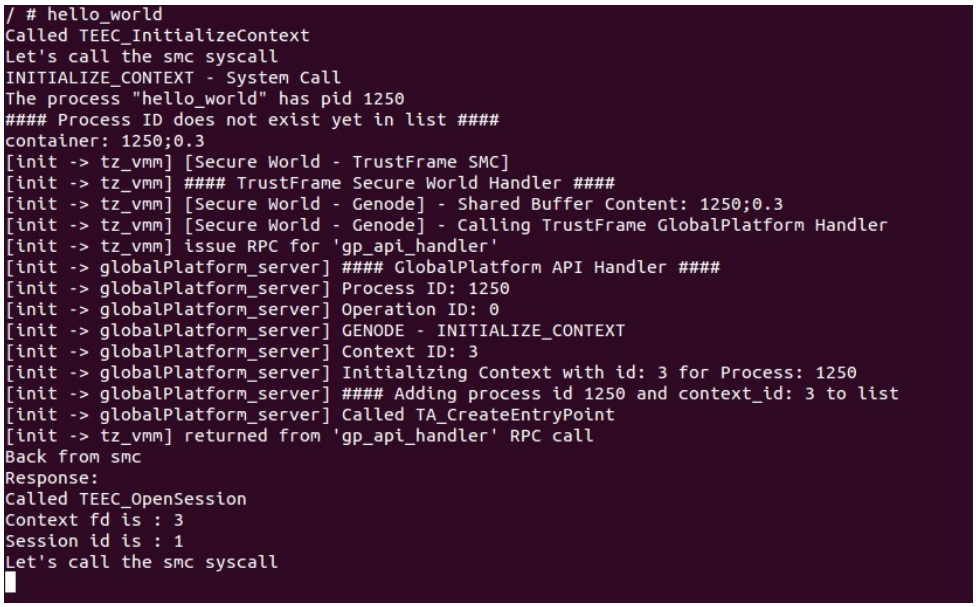

Figure 5.9: Hello World starting to execute.

meaning that they must be taken care of before the invocation occurs. Only when the client application issues a *TEEC_CloseSession* or a *TEEC_FinalizeContext* it does not require the same verifications mentioned above, since these functions have no return. Ideally, these patterns should be automatized to ease the workload of the developer. Although we identified these repetitive patterns and identified the possibility of automatizing these replications of source code, we did not implement such mechanisms, due to other priorities.

Our example executes three different trusted services, each called after the other. The first service executed is the sample we encountered, which receives a certain value from the normal world CA and increments it, returning the new value back to the CA. The second service is similar to the first, but instead of a value, returns a date. The third service uses a different data type, instead of the previous value type. This service requires the CA to prepare a data buffer, containing the desired data, which in this example is a string. This data buffer is then sent the same way to the secure world, where the trusted service receives the string and responds with another string, similar to a TCP/IP acknowledge response.

Figure 5.9 illustrates this example beginning its execution. In this figure, we can see that the first function called is *TEEC_InitializeContext*, which calls the system call and triggers the *smc*. Afterwards, we see the *init* component of Genode redirect to the *tz_vmm* component, where our secure world handler retrieves the data from the shared memory region and displays it, as can be seen. In this step, not much data is sent, only the process identifier, the operation identified (InitializeContext) and the context identifier. After this step, our GlobalPlatform API handler is called, using the Genode RPC interface, which then executes the secure world counterpart function of *TEEC_InitializeContext*, *TA_CreateEntryPoint*, returning afterwards to the normal world to continue the execution of the client application.

5.4 SoC Details

In this section we will provide some details concerning the SoC used for this work, namely the NXP i.MX53 QSB. These minor details concern the central security unit, the multi-master multi-memory interface and the TrustZone interrupt controller, which are peripherals from the development board used. We will explore, in some detail, how these components were used in our system.

5.4.1 NXP i.MX53 QSB

During the initial development phase of our work, we learned that our NXP i.MX53 QSB contained components which would require our attention. These component, as we learned, were specific to our board and would need to be configured properly if we were to successfully boot Linux in the normal world. We began by the central security unit (CSU), which is a custom TrustZone protection controller, responsible for setting comprehensive security policies within the i.MX53 platform, and for sharing security information between the various security modules. The CSU allows the assignment of peripherals access to the secure or normal worlds using corresponding configuration bits, by setting the security control registers (SCR) during boot time, through the advanced high assurance boot (A-HAB) code and locked after, to prevent further writing. The high assurance boot code is part of the secure boot ROM.

Not much information is provided concerning the CSU, as well as the other components mentioned above, which turned out to be quite a difficult task. We searched the NXP community forums for answers and after some time we found one for our problem. We immediately started to configure the CSU, by setting the CSL0 to CSL31 registers to 0x00FF00FF, as was stated in the forums. After this step, we still needed to configure the multi-master multi-memory interface (M4IF) and the TrustZone interrupt controller (TZIC), and so we investigated how to properly configure these components. The M4IF is part of the DDR memory controller and enables DDR RAM resources to be masked, allowing certain memory regions to be granted exclusive access by the secure world. We followed some tips found in many forums, on how to configure the M4IF to allow memory partition between both worlds, but were not sure we succeeded since, at that time, our Linux boot would hang. We believed that this was not due to these configurations, but was instead from the Linux kernel itself, who was probably executing some instructions that should not be executed in the normal world. The remaining component that needed to be configured was the TZIC, which is a TrustZone enabled interrupt controller with an AXI interface. This peripheral collects interrupt requests and routes them to the ARM core. It allows a complete control over every interrupt connected to the controller, like the security or priority registers functionality. Each interrupt can be configured as a normal or a secure interrupt, through restrictions like non-secure read and write transactions to only those interrupts configured as non-secure and secure read and write transactions to all secure interrupts. To configure the TZIC, we followed the specifications found in [44].

Although we did configure all these components, we ended up not using them, since we opted to use Genode later on. Since Genode already contained these configurations, plus the Linux modifications necessary to boot in the normal world, we left this part of our work and used Genode's, who presents similar configurations to ours, in terms of these components.

5.5 Summary

In this chapter we have been through the design and implementation of TrustFrame. We began by providing details over the development path for this work, explaining the main challenges encountered, along with some decisions made along the way.

Afterwards, we started by exploring the implementation details concerning the normal world, namely the Linux OS, the custom system call implemented, the GlobalPlatform TEE Client API and some security concerns. We continued with the secure world implementation details, where we provided information details on the modifications made to the Genode TrustZone component *tz_vmm*, followed by details on our secure world module, where we implemented our GlobalPlatform API handler, along the TEE Internal API. Furthermore, we provided more details on how trusted services where implemented and were executed by the client application, followed by an explanation on the Hello World example provided in this work, as proof of concept.

Finally, we explored some details concerning the development board used for this work. In the next chapter we present the experimental evaluation made for our prototype, along with the respective results.

6 Evaluation

In the previous chapter we presented the developed solution, along with the implementation details for each execution world. In this chapter we provide our evaluation methodology and setup, as well as the results for our prototype.

6.1 Methodology

Since our solution uses ARM TrustZone, we sought to measure the execution times in each world, along with time measures of our system call, in order to understand the impact it has on the performance of the overall system. We divided these measurements into different sets:

Execution of services in the normal world. In this test we limit our measurements to the normal world. We execute the client application, but instead of invoking the TEE Client API, consequently calling our system call, we simply execute the services solely in the normal world. This allows us to retrieve the performance results for the three services in the normal world only, which along the overall results, allows us to separate the impact of running the same source code in each world.

Execution of normal world services with secure world invocations. We continued to measure the three services (Increment, Print Date and Copy String) in the normal world. The difference in this test is the invocation the GlobalPlatform TEE Client API functions that will always be executed in every client application, namely the pair `TEEC InitializeContext` and `TEEC FinalizeContext` functions and the pair `TEEC OpenSession` and `TEEC CloseSession` functions. Since every client application must invoke these functions, we measured what impact would the invocation of those functions have, along with the three services, running solely in the normal world. This way, we can measure the execution performance of a given service, along with the data manipulation of our system in the secure world.

Execution of secure world services. Afterwards, instead of executing the service in the normal world, like the previous test, we execute it in the secure world, by invoking the `TEEC InvokeCommand` function in the client application. With the performance measurement of this test, plus the performance results of the previous test, we can prove that, as expected, each service takes the same time to execute in the secure world, as in the normal world, without any meaningful impact on performance.

Execution of the entire system. Finally, we measured the performance of the entire system, from point to point. Starting from the beginning of the client application execution in the normal world, consequently calling our system call, executing commands in the secure world and receiving the response. Each service is measured separately, along with the invocation of the GlobalPlatform API in both worlds. This allows us to retrieve overall performance details, which we use with the previous set

of tests, to retrieve more specific details of our system performance, like the overhead introduced by our system. Along with the system overhead, we look to measure the biggest time contribution to the overall performance results.

As for the objective of these tests setups, we aimed at measuring the execution time of:

Every service in the normal world, with no invocations to the secure world.

Each service in the normal world, plus the invocation of the GlobalPlatform API functions to the secure world, without accounting for the overhead introduced by setting and getting the updated data.

The invocation of the GlobalPlatform API functions to the secure world, with the desired service being called in the secure world, without accounting for the overhead introduced by setting and getting the updated data.

Each service in the secure world, plus the invocation costs and the overhead introduced by setting and getting the updated data.

6.2 Testing Setup

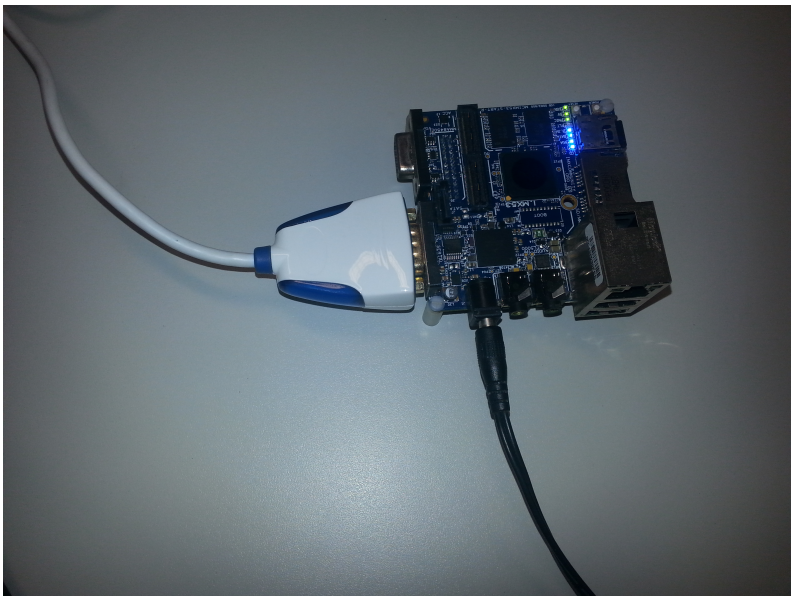

Figure 6.1: NXP i.MX53 QSB connected to a laptop.

The evaluation setup consisted in a single laptop connected to the NXP i.MX53 QSB (as seen in Figure 6.1), through an UART cable. The NXP i.MX53 QSB contains an ARM Cortex-A8, which is a 32-bit processor with processing speeds from 600 MHz to 1,2GHz. This processor belongs to the superscalar processor family, where the CPU implements a form of parallelism known as instruction-level parallelism, increasing the number of instructions per cycle, i.e., the number of instructions that are executed in a unit of time, and therefore, the throughput. This parallelism allows the processor to achieve high ratings in simple benchmark tests like the Dhrystone MIPS (Million Instructions per Second), or DMIPS, per MHz test, where the ARM Cortex-A8 may achieve the rating of 2 DMIPS. In

this test the processor executes simple arithmetic instructions. Although it was developed in 1984, it is still a common used computing benchmark test nowadays. This rating show that this processor is ideal for straight forward algorithms, with a low number of branches, data loads or data processing, thereby maximizing the number of DMIPS it may process, otherwise it may struggle to achieve the 2 DMIPS.

The solution developed for this work was executed in the NXP i.MX53 QSB, by booting a system image located in the MicroSD inserted in the development board. Through the UART cable, the output from the developed board was received in a terminal console, in the laptop. The evaluation tests were executed in this terminal console through the use of scripts, which ran our Hello World example multiple times, retrieving the performance details after each execution. To allow the execution of the sets of tests, described above, several system images were deployed into the MicroSD card, separately for each test, each containing the necessary modifications for the test at hand.

To allow the retrieval of performance results with great precision, each test was executed as a CPU intensive task, running in 100K cycles. Within the cycle, the tested service was called ten times, to further reduce the impact of calling the timing function, reaching a total execution of 1M cycles. The timing function used was `clock_gettime(CLOCK_MONOTONIC,...)`, which has nanosecond precision, which is the granularity level that we needed. This allowed us to calculate the average execution performance with the desired granularity.

6.3 System Overhead

In this section we will discuss the overhead introduced by our system implementation. We will explore the main contributions to this overhead, and why they exist.

One of the purposes for the tests described above was the measurement of the overhead introduced by our system. Ideally, we would like one service to take the same execution time, regardless of the execution world where it executes, with minimal impact from the TEE. In our system, the data retrieved from the normal world must first pass through the *tz_vmm* component of Genode and then pass to our GlobalPlatform module. In this module the data is manipulated and stored, being retrieved by the *tz_vmm* component later on. This data manipulation between the *tz_vmm* component and the implemented GlobalPlatform module, along with the data manipulation within the module, adds an overhead to the final execution time, as expected. The GlobalPlatform specifications contain three different ways to handle parameters, and in our system we implemented two of the three. We handle parameters either by value or by a data buffer. For each type of handling, the final overhead introduced will vary, since the manipulation of the data buffer has an increased cost to the overall performance, compared to the manipulation by value.

The overhead introduced by the system was calculated by removing the service execution time average from the overall time average. Throughout the testing of the system, we noticed that the invocation of the `TEEC InitializeContext` and `TEEC FinalizeContext` functions had minimal impact on overall performance, with estimates of two nanoseconds. The same was observed from the invocation of both the `TEEC OpenSession` and `TEEC CloseSession` functions, with averages between one and two nanoseconds. These small execution times are lower than expected, being originated from the error

margin in the test results and possibly by errors in the time calculation measurements. The biggest time contribution belongs to the `TEEC InvokeCommand` function, mainly due to the fact that when this function is called and all the data is sent, it must be parsed in the secure world, to be assigned and used accordingly. The measurements for each of these functions already contain the execution costs of the corresponding functions of the secure world, i.e., from the TEE Internal API, due to the small added cost implied. More details will be provided further ahead.

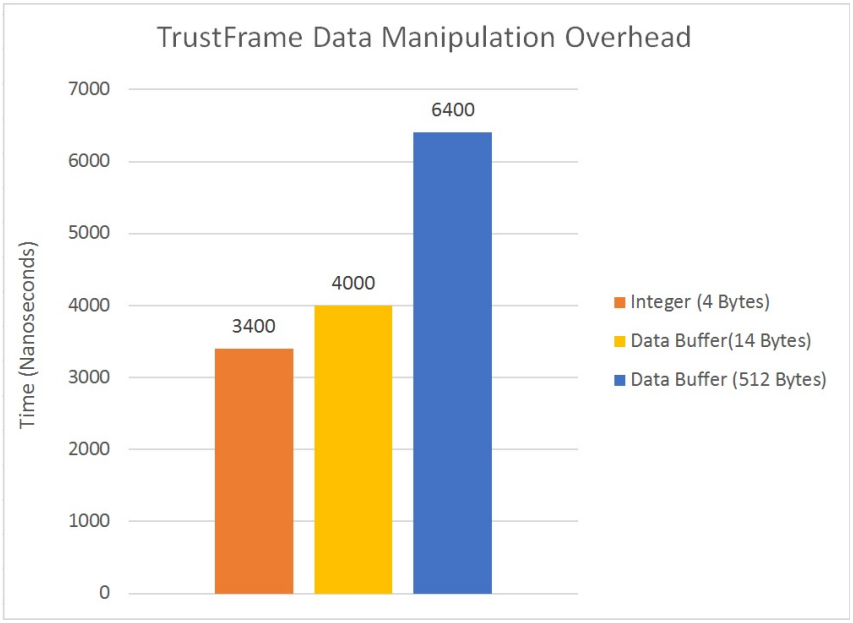

Figure 6.2: Time overhead added for different amounts of manipulated data.

For the manipulation by value, as used when executing the increment value service, we observed an average overhead time of 3400 nanoseconds. For the data buffer manipulation, the overhead varied based on the amount of data bytes being manipulated. To measure the variation, we tested with different amounts of bytes being manipulated. The first two measurements contained 8 and 14 bytes of data, presenting an average overhead of 4000 nanoseconds, while the third measurement contained 512 bytes of data being manipulated, presenting an average overhead of 6300 nanoseconds. The biggest time contribution for these overhead results is the cost of updating and retrieving the GlobalPlatform parameters, manipulated by the services, along with the data exchange between the *tz_vmm* component and our GlobalPlatform module. Each service must call the appropriate parameter updater, based on the type of data the service manipulated, i.e., by value, like in our increment service, or through the data buffer like our string copy service. Before setting and retrieving the updated data, a cycle is executed, based on the normal world process and session identifier, along with the secure world session identifier, assigned when a new session is opened. This cycle is responsible for providing a considerable contribution to the final overhead. If the service manipulated the data buffer parameter, a small amount of time is added to the overhead, due to the use of an extra *memcpy* function, that is only used for this type of parameters.

Figure 6.2, contains the introduced overhead for the different types of data manipulation used in the services, described above. As seen in the figure, the time cost per byte decreases when the amount of bytes being used increases, i.e., the difference in manipulating 4 bytes and 14 bytes has an average time cost of 600 nanoseconds, while the number of bytes increased by 10 bytes, resulting in a cost of 60

nanoseconds per byte. When the amount of bytes increases to 512 bytes, the average time cost added is 2400 nanoseconds, when manipulating additional 498 bytes, resulting in a cost of approximately 4,8 nanoseconds per byte, an approximate increase in performance by 92%. These are very good results, which allow us to conclude that data manipulation with data buffers have a much better cost per byte racio than data manipulation by integer value. This is also enforced by the data handling of TrustFrame, since data manipulation through the data buffer is handled using the *memcpy* function more often, which is one of the most efficient ways to copy bytes in memory, since it is highly optimized for such tasks.

The overhead added by our system was already expected. This is due to the fact that, instead of only copying the same code from the normal world to the secure world, we also added more code that must execute before and after the service executes. This is required, not only due to our implementation of the GlobalPlatform specifications, but also due to the requirements of Genode's RPC service. Even though the system overhead brings the nanosecond execution time of both increment value service and print date service to the microsecond execution time, using the system, the added overhead is imperceptible to the developer, since the time scale is still at a few microseconds, which is still very fast. In the data buffer manipulation overhead, although the amount of bytes changed from 14 bytes to 512 bytes, around 37 times more, the overhead variation was roughly 1,6 times, which is insignificant at this scale of time.

6.4 Service Comparison

In this section we will explore in more details for each service. We will provide further details on the execution of each service, specifically the execution isolated from the remaining workload executed by both our module and the *tz_vmm* component, in order to understand the impact of the GlobalPlatform API invocations made by the client application. One of the purposes for these measurements was to confirm that it takes the same execution time to run a certain code in the secure world, as it would in the normal world, since the hardware is the same, with the same CPU processing the same instructions, in both worlds.

6.4.1 Increment Service

This service consisted into incrementing the received value. Since this is a simple task, it was executed several times, as stated previously.

We started by measuring the service running isolated in the normal world, which required 19 nanoseconds, on average, to execute. We then measured the costs of invoking the GlobalPlatform API functions, at the same time the service was being executed. Since the `TEEC InitializeContext` and `TEEC FinalizeContext` functions and the `TEEC OpenSession` and `TEEC CloseSession` functions must always execute one time only, we measured their invocation cost, along with the increment service, which resulted in an additional cost of three nanoseconds, on average, to the previous 19 nanoseconds, totalling 22 nanoseconds. After this test, the only function left to be measured was the `TEEC InvokeCommand`. We then measured the invocation of all the GlobalPlatform API functions, first while running the service in the secure world and then without running the service at all. This allowed us to

Figure 6.3: Increment service performance results in both worlds.

retrieve the just the cost of invoking the `TEEC InvokeCommand` function, where we obtained and average of 126 nanoseconds, totalling an average of 148 nanoseconds for this service.

Afterwards, we measured the service executing solely in the secure world, where we isolated the cost of invoking the GlobalPlatform API functions. We obtained the same time average of three nanoseconds, for the all the GlobalPlatform API functions, excluding the `TEEC InvokeCommand`. For the command invocation, the measured an average of 129 nanoseconds and for the service in the secure world we measured an average of 21 nanoseconds, totalling an average of 153 nanoseconds, a slight difference from the previous test, due to small error margins. An overview of these results can be seen in Figure 6.3, where we partition the time costs described above.

Although the system forces a delay around 130 nanoseconds in the service execution, due to the execution of the GlobalPlatform API functions, this impact is only highlighted since this service executes at the nanoseconds scale. If this service executed at the microsecond scale or above, the delay impact would be negligible.

Finally, after measuring all the stages of the service invocation from the client application, to the execution in the secure world, we could compare the times of just executing the service in both world, where we ended up confirming that it takes practically the same time, on average, to execute, taking into account the small error margin observed.

6.4.2 Print Date Service

This service consisted into storing the received parameter values into a date structure, printing a date afterwards. Since this service uses the same parameter manipulation as the increment service, the difference in performance results is minimal.

We executed the same steps, used in the previous test, starting by measuring the service executing solely on the normal world, where we obtained an average time of 21 nanoseconds. Since the invocation of the GlobalPlatform API functions necessary for the initialization, finalization, opening a new session and closing that session are static, between services, i.e., are always the same, their times average is always around three nanoseconds. For that reason we measured the total invocation costs, without executing the service, and removed the average time for the GlobalPlatform API functions mentioned above, resulting in the average time of the `TEEC InvokeCommand` function of 130 nanoseconds, totalling 151 nanoseconds, on average, for this service. This result is practically the same time average for the increment service, since the data flow is the same.

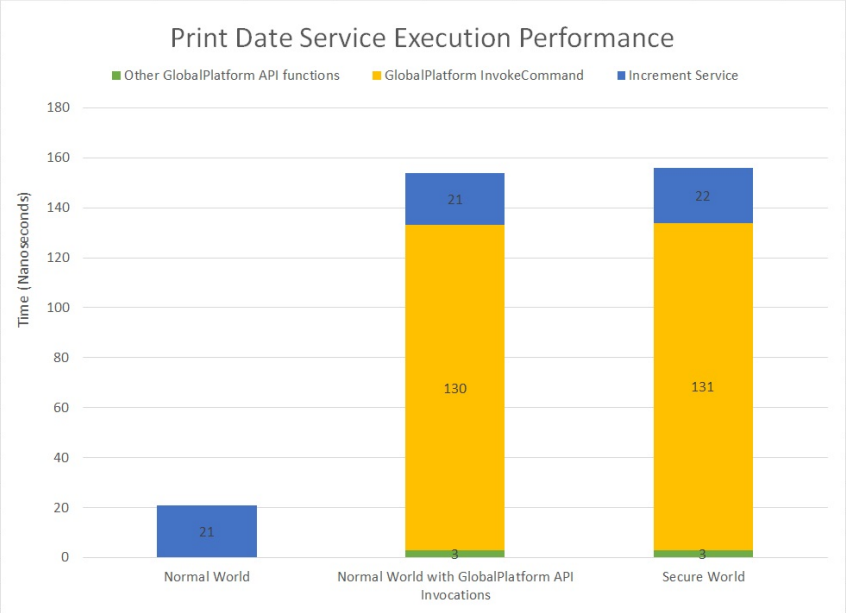

Figure 6.4: Print date service performance results in both worlds.

When measuring the service executing in the secure world, we obtained the same three nanoseconds for the GlobalPlatform API functions, excluding the `TEEC InvokeCommand`. Thereafter, we measured the invocation cost, resulting in an execution time average of 131 nanoseconds, followed by a time average of 22 nanoseconds for the service in the secure world. Figure 6.4, contains an overview of the results mentioned above.

Just like in the previous service, the delay of approximately 133 nanoseconds is due to the execution of the GlobalPlatform API functions. It would also be negligible if the service execution performance was at the microsecond scale or above.

After the measurements we could compare the execution times of the service in both worlds. Again,

we confirmed that the differences, between executing the same code in the secure world or in the normal world are minimal, where the small difference is attributed to small error margins.

6.4.3 String Copy Service

This service consisted into copying a received string, which could be used in the secure world, for any means, and copying a response string, to be sent back to the normal world. This services was developed as a proof of concept, for the data manipulation with the data buffer parameter from the GlobalPlatform API specifications.

For this service we executed the same tests, but using different payloads, to ascertain the performance costs of increasing the amount of data manipulated in the secure world. We measured the service using 8 bytes, 14 bytes and 512 bytes. The first two payloads, derive from copying the strings ”Hello” and ”Hello World”, plus the response string ”ACK”, while the latter, derives from the amount of bytes being copied, where we changed from the length of the string and forced the copy of 512 bytes. Since both strings are copied, one after the other, we considered the payload amount as the sum of the strings being copied. The major differences between this service and the remaining two are that this service manipulates data from a data buffer parameter, specified by the GlobalPlatform API standards. We used the *memcpy* function more often, as a solution, which brought an increase in the time average, compared to the previous services.

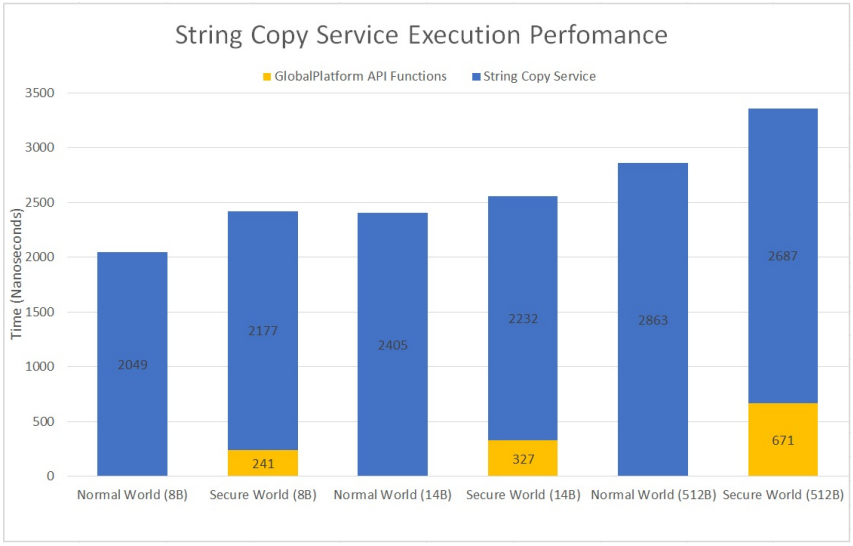

Figure 6.5: String copy service performance results in the normal worlds.

Figure 6.5, contains an overview of these results. In terms of the service execution isolated in the normal world, using the service to copy a total of 8 bytes resulted in an average execution time of 2049 nanoseconds. The same service copying a total of 14 bytes only took around 1,17 times more, about 2405 nanoseconds, although the amount of data increased 1,75 times. As for the final test, copying 512 bytes, the average time was 2863 nanoseconds, which is an increase of roughly 1,19 times the execution time of the same service copying 14 bytes of data, even though the amount of data increased about 36.5 times.

When executing the same service in the secure world, using the service to copy 8 bytes of data resulted in an average time of 241 nanoseconds for the invocation of the GlobalPlatform API functions and an average of 2177 nanoseconds, totalling 2418 nanoseconds, on average. When using 14 bytes of data with this service, the execution time obtained, from the invocation of the GlobalPlatform API functions, was 327 nanoseconds and for the service was 2232, totalling 2559 nanoseconds, on average. Finally, when using 512 bytes of data with this service, the average execution time for the invocation of the GlobalPlatform API functions was 671, while the service only took 2687 nanoseconds to execute, totalling 3358 nanoseconds, on average.

The difference in the execution time of the service isolated, is due to the observed standard deviation of 190 nanoseconds, over the service's execution time, which slightly changes the ratios calculated above. The time difference between executing the service, using 14 bytes and 8 bytes, only varies roughly 1,05 times, while the time difference increases 1,31 times, when using 512 bytes of data. In this service, as opposed to the previous services, the service execution delay introduced by the excution of the GlobalPlatform API functions has a small impact on the overall performance, since this service executes at a higher time scale. This impact would be even smaller if the service would execute during longer time periods, as mentioned previously.

6.4.4 Service Overview

As stated above, the system introduces a certain overhead, based on the type of data parameters being handled and, in case of the data buffer parameter, amount of bytes being manipulated. After analyzing each service isolated form the remaining tasks performed by our module, along the *tz_vmm* component of Genode, we analyze the total execution time for each service.

The total execution times for the increment service and print date service remain with minimal differences, due to the fact that they manipulate the received values from the parameter structure of the GlobalPlatform specifications in the same way. For the increment service, the average total time is 3528 nanoseconds, while the average total time for the print date service is 3558 nanoseconds. As for the string copy service, the introduced overhead varies, based on the amount of data being manipulated. For the manipulation of 8 bytes of data, the average total time is 6272 nanoseconds, while for the manipulation of 14 bytes, the total time slightly increases to 6375 nanoseconds, on average. Finally, for the last scenario, the total time of the string copy service, while manipulating 512 bytes of data is 9711 nanoseconds, on average.

The time cost per byte decreases with the amount of bytes being manipulated. This increase in efficiency is more noticeable in the string copy service total results. Here, we observe that the difference in manipulating 8 bytes and 14 bytes is around 103 nanoseconds, which results in a time cost per byte of 17 nanoseconds, approximately. If compared with the total execution using 512 bytes, we obtain a difference around 3336 nanoseconds for an additional 498 bytes being manipulated. This results in a time cost per byte of just 6,7 nanoseconds, approximately, which again illustrates the efficiency of the system and how it can scale with greater amounts of bytes being manipulated in the service.

Although the amount of data manipulated varied considerably between each service, as well as the way our GlobalPlatform module handled the data, the difference in the total time, did not escalate

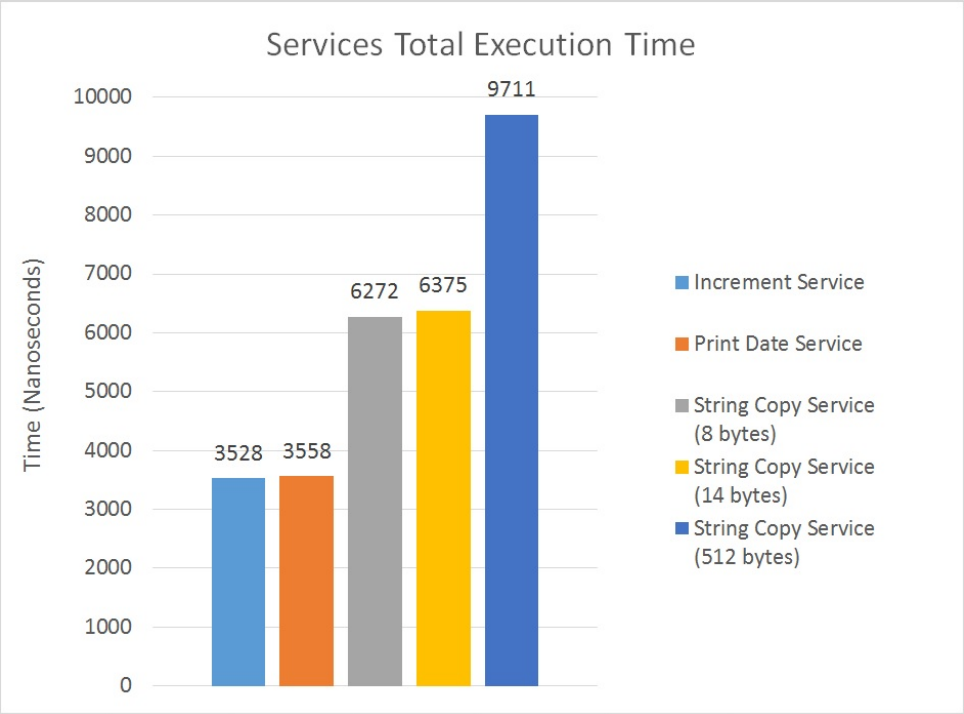

Figure 6.6: Total execution performance for the provided services.

proportionally to the increase in the amount of data being handled. These results show that our system provides very good scalability, even if larger amounts of data are used within our module, as seen by the results of varying from a simple integer of 4 bytes, being incremented, to a 512 bytes string, being copied. Figure 6.6 contains the overview of the total execution time for each service.

6.5 Qualitative Analysis

Our system combines the use of Genode with the GlobalPlatform API standards. Being based on Genode, a highly sought development framework for TrustZone related projects, and capable of using the API devised by GlobalPlatform, that is becoming increasingly used in new development frameworks, our system allows applications developed in other frameworks, compatible with the GlobalPlatform API standards, to execute in our environment without having to make further modifications. Besides porting applications from other frameworks, developers can also port their trusted services, provided they adapt them to any restraint that could be imposed by the use of Genode.

Since our system is based on Genode, libraries ported for Genode can also be used in our system, for new solutions like trusted services. Currently there are already many libraries that are supported by Genode, like SQLite or OpenSSL, among others, offering new possibilities for new services to be developed and used in our GlobalPlatform module.

By looking at the services we already provide (see Figures 6.7 and 6.8), developers that want to implement their own service, may adapt their solution to meet the requirements for our system, without great effort. Besides the difficulties adherent from the logic that the developer's implemented solution

may require, not many more difficulties are foreseen, since there are examples from which the developers may base themselves, as well as the Genode community for Genode related problems.

```
    /* Print Date Service Invocation */
  memset(&op, 0, sizeof(op));
  op.paramTypes = TEEC_PARAM_TYPES(TEEC_VALUE_INOUT, TEEC_NONE,
              TEEC_NONE, TEEC_NONE);
  op.params[0].value.a = 24;
  op.params[0].value.b = 12;
  printf("Invoking TA to print the date after %d-%d-2016\n",
          op.params[0].value.a, op.params[0].value.b);
  res = TEEC_InvokeCommand(&sess, TA_FUNC_PRINT_CMD_DATE, &op,
           &err_origin);
  if (res != TEEC_SUCCESS)
     errx(1, "TEEC_InvokeCommand failed with code 0x%x origin 0x%x",
        res, err_origin);
  printf("The TA date is %d-%d-2016\n", op.params[0].value.a, op.params[0].value.b);
```

Figure 6.7: Example of an invocation of a trusted service, by the client application.

```
    /* Print Date Trusted Service */
    TEE_Result printNextDate(uint32_t param_types, TEE_Param params[4])
{

  if(verifyParamTypesInout(param_types) != TEE_ERROR_BAD_PARAMETERS){
    date d;
    d.day = params[0].value.a + 1;
    d.month = params[0].value.b;
    /* No more than two value parameters defined in TEE_Param structure */
    d.year = 2016;
    setUserParameters(d.day, d.month);
    return TEE_SUCCESS;
   }
   return TEE_ERROR_BAD_PARAMETERS;
}
```

Figure 6.8: Example of a trusted service.

6.6 Summary

In this chapter we presented the experimental evaluation made for TrustFrame, along with the obtained results. The next chapter concludes this thesis, by presenting the conclusions regarding the work developed, introducing as well some directions in terms of future work.

7 Conclusions

7.1 Conclusions

As the value and confidentiality of sensitive data stored in mobile devices increases, so does the necessity of better security assurance. However, providing such assurance is not an easy task, since most common security solutions rely on complex systems, with many lines of code, potentially containing bugs, waiting to be explored, resulting in loss of theft of data, possibly harmful for the user.

Several works have been presented, in the recent years, offering new solutions, based on novel technologies like ARM TrustZone. In these solutions, sensitive code may be executed in an isolated environment, separated from the main operating system, protecting the confidentiality and integrity of code deemed critical.

Nonetheless, it is still complicated to develop new solutions that use ARM TrustZone, due to several difficulties, like the configuration of the execution environment, the choosing appropriate TrustZone-aware hardware, lack of good hardware documentation, constant evolution of development boards and high complexity of TrustZone security systems.

Some of the presented works related to ARM TrustZone technology focus on providing small, but trusted, services that run in the secure world, isolated from a traditional operating system running in the normal world. These services range from authentication mechanisms, one-time passwords, use of cryptology or virtualization features to reduce their attack surface by minimizing the trusted computing base. Other solutions led to the creation of development frameworks, capable of exploring TrustZone features. However, there are still difficulties in using such platforms, due to the lack of support or good documentation, to the lack of availability of their source code.

In this work, we proposed the development of TrustFrame, a toolkit capable of allowing the development of new TrustZone-based solutions, that could ease the difficulties of developers and serve as a basis for new TrustZone projects in the future. We based our solution on Genode, an existing development framework capable of using ARM TrustZone, where we combined the use of a supported and easily available framework, with the API standards devised by GlobalPlatform, an organization that has been dedicating itself into the development of standardized specifications for trusted execution environments, where trusted solutions execute. Our system was tested using three different types of services, as a proof of concept, with different complexity levels among them.

By analyzing our evaluation results, we realized that, although our system introduced some overhead in the overall performance, that impact is still minimal and insignificant to developers and their solutions. The devised services remained with high speed performance results, while running in the secure world and using the GlobalPlatform API specifications.

7.2 Future Work

As for future work, the developed solution enables many possibilities. These include the following:

Extend the GlobalPlatform API. Since not all the API specifications were implemented during this work, it is still possible to extend the current implementation and add new functionalities. The GlobalPlatform specifications account for the use of cryptography, arithmetic, trusted storage, as well as other APIs, which were not implemented in this work, leaving room for future work.

Enhance the use of Genode. Since Genode is a complex system, its potential was not fully explored in this work, leaving room for some system limitations, like data exchange in the RPC service. Such limitations could be optimized, further enhancing the capabilities that this toolkit has to offer.

Optimize the GlobalPlatform module. During the evaluation tests, it was noted that the system introduced an overhead to the overall execution of a service. Optimizing the data flow and reducing the copy of data, within this module, would surely have an increase in the overall performance results.

Code production automation. During the development of our solution, we noticed that the client application contained duplicated source code, for example, when calling the `TEEC InvokeCommand` function. Automation of the generation of duplicated code, similar to the RPC systems, would benefit future development projects, ultimately aiding the developer.

Increase compatibility and portability of the system. By using Genode, this platform is a candidate for other development boards, since Genode itself supports other development boards. Nonetheless, the *tz_vmm* demo, which served as a basis for this work is not prepared to execute in all those development boards, since it was only adapted to a few different SoCs. Extending the support to other development boards would surely reach broader audiences and increase interest in this type of work. Although we already support Linux as a rich OS, supporting more REE OSes, like Android, would open new possibilities for more future work. Finally, use of a new toolchain, compatible with this work, could provide better access to the toolchain and further support. This way, issues, like the ones described earlier, that led to compilation and availability problems, making it hard to find the appropriate toolchain, would be mitigated.

Bibliography

1. Educause. 7 Things You Should Know About Mobile Security. pages 1–2, 2011.
2. Check Point Software Technologies. The Impact Of Mobile Devices On Information Security: A Survey Of IT Professionals. (June):1–12, 2013.
3. Arm. ARM Security Technology. Building a Secure System using TrustZone Technology. page 108, 2009.
4. Samsung Knox Platform. White Paper: An Overview of Samsung KNOX. (June), 2015.
5. Norman Feske. Genode Operating System Framework 16.05.
6. NXP Semiconductor. i.MX53 Quick Start-R Board.
7. ARM Limited. ARMv7-A and ARMv7-R edition. page 2734, 2012.
8. ARM Limited. ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile. 2014.
9. John Goodacre and Program Management. Technology Preview : The ARMv8 Architecture. (November):1–10, 2011.
10. Johannes Winter. Trusted Computing Building Blocks for Embedded Linux-based ARM Trustzone Platforms. *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, pages 21–30, 2008.
11. T Alves and D Felton. Trustzone: Integrated hardware and software security. *ARM white paper*, 3(4):18–24, 2004.
12. GlobalPlatform. `http://globalplatform.org/`, 2016. [Online; accessed 17-October-2016].
13. Trusted Computing Group. `http://www.trustedcomputinggroup.org/`, 2016. [Online; accessed 17-October-2016].
14. GlobalPlatform. GlobalPlatform Device Specifications. `http://www.globalplatform.org/specificationsdevice.asp`, 2016. [Online; accessed 17-October-2016].
15. GlobalPlatform. GlobalPlatform Device Technology TEE Client API Specification. (July):1–58, 2010.
16. GlobalPlatform. GlobalPlatform Device Technology TEE Internal API Specification. (June):1–202, 2011.
17. Xiaolei Li, Hong Hu, Guangdong Bai, Yaoqi Jia, Zhenkai Liang, and P. Saxena. DroidVault: A Trusted Data Vault for Android Devices. pages 29–38, 2014.

18. He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. TrustOTP : Transforming Smartphones into Secure One-Time Password Tokens. *Ccs*, pages 976–988, 2015.

19. Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-kang Chu, and Tieyan Li. Building trusted path on untrusted device drivers for mobile devices. *Proceedings of 5th Asia-Pacific Workshop on Systems - APSys '14*, (JUNE):1–7, 2014.

20. Ahmed M Azab. Hypervision Across Worlds : Real-time Kernel Protection from the ARM TrustZone Secure World. *Ccs*, pages 90–102, 2014.

21. Emre Koyuncu. vTZ: A Case for TrustZone Virtualization Zhichao. page 4503.

22. He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. Reliable and Trustworthy Memory Acquisition on Smartphones. 10(12):2547–2561, 2015.

23. He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. TrustICE : Hardware-assisted Isolated Computing Environments on Mobile Devices.

24. Bo Yang, Kang Yang, Yu Qin, Zhenfeng Zhang, and Dengguo Feng. DAA-TZ: An Efficient DAA Scheme for Mobile Devices using ARM TrustZone. pages 1–22, 2015.

25. Wenhao Li, Haibo Li, Haibo Chen, and Yubin Xia. AdAttester : Secure Online Mobile Advertisement Attestation Using TrustZone. *MobiSys*, pages 75–88, 2015.

26. Jinsoo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent Byunghoon Kang. SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment. *Proceedings 2015 Network and Distributed System Security Symposium*, (February):8–11, 2015.

27. Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using ARM trustzone to build a trusted language runtime for mobile applications. *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems - ASPLOS '14*, (i):67–80, 2014.

28. Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Trusted language runtime (TLR): enabling trusted applications on smartphones. *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile)*, pages 21–26, 2011.

29. Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. SPROBES: Enforcing Kernel Code Integrity on the TrustZone Architecture. *In Proceedings of the 3rd IEEE Mobile Security Technologies Workshop (MoST)*, 2014.

30. Claudio Marforio, Nikolaos Karapanos, and Claudio Soriente. Smartphones as Practical and Secure Location Verification Tokens for Payments. *NDSS 2014 (21st Network and Distributed System Security Symposium)*, (February):23–26, 2014.

31. Ferdinand Brasser, Daeyoung Kim, and Christopher Liebchen. Regulating Smart Personal Devices in Restricted Spaces. (July), 2015.

32. Javier González. *Operating System Support for Run-Time Security with a Trusted Execution Environment*. 2015.

33. Christoffer Dall and Jason Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. (July):333–347, 2014.

34. N. Asokan, Jan Erik Ekberg, and Kari Kostiainen. The untapped potential of trusted execution environments on mobile devices. *IEEE Security And Privacy*, 12(4):293–294, 2013.

35. Brian Mcgillion, Tanel Dettenborn, Thomas Nyman, and N Asokan. Open-TEE - An Open Virtual Trusted Execution Environment. 2015.

36. Markus Katzenberger. <t base Trusted Application Development. 2014.

37. Linaro and STMicroelectronics. OP-TEE, open-source security framework, http://www.linaro.org/blog/core-dump/op-tee-open-source-security-mass-market/. 2015.

38. Sierraware. SierraTEE an ARM TrustZone and ARM Hypervisor Open Source Software, http://www.openvirtualization.org/.

39. Sierraware. SierraTEE Overview.

40. TrustKernel. T6: Secure OS and TEE. pages 1–6, 2015.

41. Andreas Fitzek. Development of an ARM TrustZone aware operating system ANDIX OS. (April), 2014.

42. Andreas Fitzek, Florian Achleitner, Johannes Winter, and Daniel Hein. The ANDIX Research OS - ARM TrustZone Meets Industrial Control Systems Security. pages 1–6.

43. Johannes Winter, Paul Wiegele, Martin Pirker, and Ronald Toegl. A Flexible Software Development and Emulation Framework for ARM TrustZone. *Trusted Systems*, 7222:1–15, 2012.

44. NXP Semiconductor. Chapter 74 TrustZone Aware Interrupt Controller (TZIC). 16:4373–4402.

A Hello World Application Example

hello_world.c

```
 #include <stdio.h>
 #include <string.h>
 #include <stdlib.h>
 #include <err.h>

/* Common includes for all applications using GP API */
#include "src/include/tee_client_api.h"

/* Specific header, concerning the Hello World Application */
#include "src/include/ta_hello_world.h"


int main(int argc, char *argv[])
{
   TEEC_Result res;
   TEEC_Context ctx;
   TEEC_Session sess;
   TEEC_Operation op;
   TEEC_UUID uuid = TA_HELLO_WORLD_UUID;
   uint32_t err_origin;


   /* Initialize a context connecting us to the TEE */
   res = TEEC_InitializeContext(NULL, &ctx);
   if (res != TEEC_SUCCESS)
      errx(1, "TEEC_InitializeContext failed with code 0x%x", res);

  memset(&sess, 0, sizeof(sess));

   // /*
   //  * Open a session to the "hello world" TA, the TA will print "hello
   //  * world!" in the log when the session is created.
   //  */
```

```
res = TEEC_OpenSession(&ctx, &sess, &uuid,
            TEEC_LOGIN_PUBLIC, NULL, NULL, &err_origin);
if (res != TEEC_SUCCESS)
   errx(1, "TEEC_Opensession failed with code 0x%x origin 0x%x",
      res, err_origin);

/*
 * Execute a function in the TA by invoking it, in this case
 * we're incrementing a number.
 *
 * The value of command ID part and how the parameters are
 * interpreted is part of the interface provided by the TA.
 */

/*  Increment Service */
memset(&op, 0, sizeof(op));
op.paramTypes = TEEC_PARAM_TYPES(TEEC_VALUE_INOUT, TEEC_NONE,
            TEEC_NONE, TEEC_NONE);
op.params[0].value.a = 42;

printf("Invoking TA to increment %d\n", op.params[0].value.a);
res = TEEC_InvokeCommand(&sess, TA_HELLO_WORLD_CMD_INC_VALUE, &op,
         &err_origin);
if (res != TEEC_SUCCESS)
   errx(1, "TEEC_InvokeCommand failed with code 0x%x origin 0x%x",
      res, err_origin);
 printf("TA incremented value to %d\n", op.params[0].value.a);

/* Print Date Service */
memset(&op, 0, sizeof(op));
op.paramTypes = TEEC_PARAM_TYPES(TEEC_VALUE_INOUT, TEEC_NONE,
            TEEC_NONE, TEEC_NONE);
op.params[0].value.a = 24;
op.params[0].value.b = 12;
printf("Invoking TA to print the date after %d-%d-2016\n",
        op.params[0].value.a, op.params[0].value.b);
res = TEEC_InvokeCommand(&sess, TA_FUNC_PRINT_CMD_DATE, &op,
         &err_origin);
if (res != TEEC_SUCCESS)
   errx(1, "TEEC_InvokeCommand failed with code 0x%x origin 0x%x",
      res, err_origin);
printf("The TA date is %d-%d-2016\n", op.params[0].value.a, op.params[0].value.b);
```

```
    /* Copy String Service */
    memset(&op, 0, sizeof(op));
    op.paramTypes = TEEC_PARAM_TYPES(TEEC_MEMREF_TEMP_INOUT, TEEC_NONE,
                TEEC_NONE, TEEC_NONE);
    char input_string_1[] = "Hello World";
    char input_string_2[] = "Hello";

    op.params[0].tmpref.buffer = &input_string_1;
    op.params[0].tmpref.size = sizeof(char) * strlen(input_string_1);

    printf("Invoking TA to print the string response to: %s\n",(char *)
        op.params[0].tmpref.buffer);
    res = TEEC_InvokeCommand(&sess, TA_FUNC_CPY_STR, &op,
              &err_origin);
    if (res != TEEC_SUCCESS)
       errx(1, "TEEC_InvokeCommand failed with code 0x%x origin 0x%x",
          res, err_origin);
    printf("TA string is %s\n",(char *) op.params[0].tmpref.buffer);

    TEEC_CloseSession(&sess);

    TEEC_FinalizeContext(&ctx);

    return 0;
}
```

B GlobalPlatform TEE Client API

tee_client_api.c

```
#include "include/tee_client_api.h"

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#include <unistd.h>
#include <errno.h>
#include <limits.h>

#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <linux/types.h>

/* print uint32_t and so on */
#include <inttypes.h>

/* How many device sequence numbers will be tried before giving up */
#define TEEC_MAX_DEV_SEQ 1

#define TEE_GEN_CAP_GP (1 << 0)/* GlobalPlatform compliant TEE */

#define SMC_FUNC_ID 366
#define MAX_SIZE 1024
#define UUID_SIZE 8

/* Operation IDs */
#define INITIALIZE_CONTEXT 0
#define OPEN_SESSION 1
#define INVOKE_COMMAND 2
#define CLOSE_SESSION 3
#define FINALIZE_CONTEXT 4
```

```
int commandSuccess = 0;
int firstTime = 0;
int good_to_go;
char args[MAX_SIZE];
TEEC_Result res;
uint32_t eorig;

int validate_input(int operation_id, char *args)
{
   char *op_id, *rest_args, *present;
   int operation;
   char dummy_buffer[MAX_SIZE];
   const char *invalid_characters = "%&";
   char *c = args;

   strncpy(dummy_buffer, args, strlen(args));

   // Extract Operation ID
   op_id = strtok_r(dummy_buffer, ".", &rest_args);
   operation = atoi(op_id);

   if(operation != INITIALIZE_CONTEXT && operation != OPEN_SESSION &&
       operation != INVOKE_COMMAND && operation != CLOSE_SESSION &&
       operation != FINALIZE_CONTEXT){
       return 0;
       }

   present = strpbrk(c, invalid_characters);
   while(present != NULL){
      printf("SMC arguments contain invalid characters!\n");
      printf("The invalid characters are: %c\n", *present);
      present = strpbrk(present + 1, invalid_characters);
      return 0;
   }
   return 1;
}

void SMC_SystemCall(int operation_id, char *args)
{
   int valid_input;

   switch(operation_id) {
```

```
case INITIALIZE_CONTEXT :
    printf("INITIALIZE_CONTEXT - System Call\n");
    valid_input = validate_input(operation_id, args);
    if(valid_input){
      sleep(1);
      good_to_go = syscall(SMC_FUNC_ID, args);
    }else{
      printf("Error - Bad input arguments\n");
      good_to_go = 0;
    }
    break;
case OPEN_SESSION :
    printf("OPEN_SESSION - System Call\n");
    if(!good_to_go){
       valid_input = validate_input(operation_id, args);
       if(valid_input){
         sleep(1);
         good_to_go = syscall(SMC_FUNC_ID, args);
       }else{
         printf("Error - Bad input arguments\n");
         good_to_go = 0;
       }
    }else{
       printf("Previous error detected...canceling operation\n");
    }
    break;
case INVOKE_COMMAND :
     printf("INVOKE_COMMAND - System Call\n");
    if(!good_to_go){
       valid_input = validate_input(operation_id, args);
       if(valid_input){
         sleep(1);
         good_to_go = syscall(SMC_FUNC_ID, args);
       }else{
         printf("Error - Bad input arguments\n");
         good_to_go = 0;
       }
    }else{
       printf("Previous error detected...canceling operation\n");
    }
    break;
case CLOSE_SESSION :
     printf("CLOSE_SESSION - System Call\n");
```

```
         if(!good_to_go){
            valid_input = validate_input(operation_id, args);
            if(valid_input){
              sleep(1);
              good_to_go = syscall(SMC_FUNC_ID, args);
            }else{
              printf("Error - Bad input arguments\n");
              good_to_go = 0;
            }
         }else{
            printf("Previous error detected...canceling operation\n");
         }
         break;
      case FINALIZE_CONTEXT :
         printf("FINALIZE_CONTEXT - System Call\n");
         if(!good_to_go){
            valid_input = validate_input(operation_id, args);
            if(valid_input){
              sleep(1);
              good_to_go = syscall(SMC_FUNC_ID, args);
            }else{
              printf("Error - Bad input arguments\n");
              good_to_go = 0;
            }
         }else{
            printf("Previous error detected...canceling operation\n");
         }
         break;
   }
}

TEEC_Result open_dev(const char *devname)
{
   int fd;
   fd = open(devname, O_RDWR|O_CREAT);
   if (fd < 0){
      return -1;
   }
   return fd;
}

TEEC_Result TEEC_InitializeContext(const char *name, TEEC_Context *ctx)
{
```

```
   printf("Called TEEC_InitializeContext\n");
   char devname[PATH_MAX];
   int fd;
   char *cur = args, * const end = args + sizeof args;
   size_t n;

   if (!ctx){
      return TEEC_ERROR_BAD_PARAMETERS;
   }
   if(firstTime){
     memset(args, 0, sizeof(args));
     firstTime = 1;
   }

   for (n = 0; n < TEEC_MAX_DEV_SEQ; n++) {
      snprintf(devname, sizeof(devname), "/dev/tee%zu", n);

      fd = open_dev(devname);

      if (fd >= 0) {

         ctx->fd = fd;
         /* Copy operation ID to args */
         cur += snprintf(cur, end-cur, "%x", INITIALIZE_CONTEXT);
         cur += snprintf(cur, end-cur, "%s", ".");

         /* Copy Context ID to args, to be sent to the Secure World */
         cur += snprintf(cur, end-cur, "%x", fd);
         printf("Let's call the smc syscall\n");

        // SMC System Call
         SMC_SystemCall(INITIALIZE_CONTEXT, args);
         printf("Back from smc\n");

         return TEEC_SUCCESS;
      }
   }
   return TEEC_ERROR_ITEM_NOT_FOUND;
}

void TEEC_FinalizeContext(TEEC_Context *ctx)
{
   printf("Called TEEC_FinalizeContext\n");
```

```
   char *cur = args, * const end = args + sizeof args;

   if (ctx){
      /* Copy operation ID to args */
      cur += snprintf(cur, end-cur, "%x", FINALIZE_CONTEXT);
      cur += snprintf(cur, end-cur, "%s", ".");

      printf("Let's call the smc syscall\n");

      // SMC System Call
      SMC_SystemCall(FINALIZE_CONTEXT, args);
      printf("Back from smc\n");

      close(ctx->fd);
   }
}

TEEC_Result TEEC_OpenSession(TEEC_Context *ctx, TEEC_Session *session,
      const TEEC_UUID *destination, uint32_t connection_method,
     const void *connection_data, TEEC_Operation *operation, uint32_t *ret_origin)
{
   printf("Called TEEC_OpenSession\n");
   session->session_id++;

   char *cur = args, * const end = args + sizeof args;
   int i;
   int firstTime = 1;

   if (!ctx || !session) {
      eorig = TEEC_ORIGIN_API;
      res = TEEC_ERROR_BAD_PARAMETERS;
      if (ret_origin)
         *ret_origin = eorig;
   }
   if (connection_method != TEEC_LOGIN_PUBLIC) {
      eorig = TEEC_ORIGIN_API;
      res = TEEC_ERROR_NOT_SUPPORTED;
      if (ret_origin)
         *ret_origin = eorig;
   }

   if(!firstTime){
```

```
    session->session_id++;
}
/* Copy operation ID to args */
cur += snprintf(cur, end-cur, "%x", OPEN_SESSION);
cur += snprintf(cur, end-cur, "%s", ".");

/* Copy UUID to args, to be sent to the Secure World */
cur += snprintf(cur, end-cur, "%08x", (unsigned long)destination->timeLow);
if (cur < end) {
    cur += snprintf(cur, end-cur, "%s", "-");
cur += snprintf(cur, end-cur, "%04x", (unsigned long)destination->timeMid);
    cur += snprintf(cur, end-cur, "%s", "-");
    cur += snprintf(cur, end-cur, "%04x", (unsigned
        long)destination->timeHiAndVersion);
    cur += snprintf(cur, end-cur, "%s", "-");
    for(i=0; i<UUID_SIZE; i++){
        cur += snprintf(cur, end-cur, "%02x", (unsigned
            long)destination->clockSeqAndNode[i]);
    }
    cur += snprintf(cur, end-cur, "%s", ",");
    /* Copy Context ID to args, to be sent to the Secure World */
    cur += snprintf(cur, end-cur, "%x", ctx->fd);
    cur += snprintf(cur, end-cur, "%s", ",");
    /* Copy Session ID to args, to be sent to the Secure World */
    cur += snprintf(cur, end-cur, "%l" PRIu32, (unsigned long)session->session_id);

    /* Copy Parameter Types */
    cur += snprintf(cur, end-cur, "%s", ",");
    if(operation == NULL){
        cur += snprintf(cur, end-cur, "%l" PRIu32, (unsigned long)TEEC_NONE);
    }else{
        cur += snprintf(cur, end-cur, "%l" PRIu32, (unsigned
            long)operation->paramTypes);
    }
}
firstTime = 0;
printf("Let's call the smc syscall\n");

// SMC System Call
SMC_SystemCall(OPEN_SESSION, args);
printf("Back from smc\n");

if(args != NULL && good_to_go){
```

```
      commandSuccess = 1;
   }

   if(commandSuccess){
      res = TEEC_SUCCESS;
   }
   return res;
}

void TEEC_CloseSession(TEEC_Session *session)
{
   printf("Called TEEC_CloseSession\n");

   if (!session)
      return;

   char *cur = args, * const end = args + sizeof args;

   /* Copy operation ID to args */
   cur += snprintf(cur, end-cur, "%x", CLOSE_SESSION);
   cur += snprintf(cur, end-cur, "%s", ".");

   /* Copy Session ID to args, to be sent to the Secure World */
   cur += snprintf(cur, end-cur, "%" PRIu32, (unsigned long)session->session_id);

   printf("Let's call the smc syscall\n");

   // SMC System Call
   SMC_SystemCall(CLOSE_SESSION, args);
   printf("Back from smc\n");
}

TEEC_Result TEEC_InvokeCommand(TEEC_Session *session, uint32_t cmd_id,
      TEEC_Operation *operation, uint32_t *error_origin)
{
      printf("Called TEEC_InvokeCommand\n");

      char *param_a, *param_b, *rest_buff, *tmpref_buffer;
      char *cur = args, * const end = args + sizeof args;
      char received_buffer[MAX_SIZE];

      if (!session) {
         eorig = TEEC_ORIGIN_API;
```

```
    res = TEEC_ERROR_BAD_PARAMETERS;
    if (error_origin){
        *error_origin = eorig;
    }
}

/* Copy operation ID to args */
cur += snprintf(cur, end-cur, "%x", INVOKE_COMMAND);
cur += snprintf(cur, end-cur, "%s", ".");

/* Copy all arguments needed by the the Secure World and send them through the
    smc */
if (cur < end) {
    /* Copy Session ID to args, to be sent to the Secure World */
    cur += snprintf(cur, end-cur, "%l" PRIu32, (unsigned
        long)session->session_id);
    cur += snprintf(cur, end-cur, "%s", ","); /* Command ID */
    cur += snprintf(cur, end-cur, "%l" PRIu32, (unsigned long)cmd_id);
    cur += snprintf(cur, end-cur, "%s", ","); /* Operation parameters type */
    cur += snprintf(cur, end-cur, "%l" PRIu32, (unsigned
        long)operation->paramTypes);

    if(cmd_id != 1){
    cur += snprintf(cur, end-cur, "%s", ","); /* Operation parameter value a */
    cur += snprintf(cur, end-cur, "%l" PRIu32, (unsigned
        long)operation->params[0].value.a);
    cur += snprintf(cur, end-cur, "%s", ","); /* Operation parameter value b */
    cur += snprintf(cur, end-cur, "%l" PRIu32, (unsigned
        long)operation->params[0].value.b);
    }else{
        cur += snprintf(cur, end-cur, "%s", ","); /* Operation memory reference
            buffer */
        cur += snprintf(cur, end-cur, "%s", (char *)
            operation->params[0].tmpref.buffer);
    }
    cur += snprintf(cur, end-cur, "%s", ",");
    cur += snprintf(cur, end-cur, "%l" PRIu32, (unsigned long)error_origin);
}
printf("Let's call the smc syscall\n");

// SMC System Call
SMC_SystemCall(INVOKE_COMMAND, args);
```

```
    printf("Back from smc\n");

    if(!good_to_go){
       // Extract parameters
       if(cmd_id != 1){
          param_a = strtok_r(args, "; ", &rest_buff);
          param_b = strtok_r(rest_buff, ". ", &tmpref_buffer);
          operation->params[0].value.a = (uint32_t)atoi(param_a);
          operation->params[0].value.b = (uint32_t)atoi(param_b);
       }else{
          tmpref_buffer = strtok_r(args, "", &rest_buff);
          strncpy(received_buffer, tmpref_buffer, MAX_SIZE - 1);
          operation->params[0].tmpref.buffer = &received_buffer;
       }

       if(param_a && param_b != NULL){
          commandSuccess = 1;
       }

       if(commandSuccess){
          res = TEEC_SUCCESS;
       }
    }

    return res;
}
```

C GlobalPlatform API Handler

main.cc

```
#include <base/component.h>
#include <base/log.h>
#include <root/component.h>
#include <globalPlatform_session/globalPlatform_session.h>
#include <base/rpc_server.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*GP includes*/
#include <globalPlatform_session/i2c.h>
#include <globalPlatform_session/tee_ta_api.h>
#include <globalPlatform_session/user_defines.h>
#include <globalPlatform_session/user_defines_utils.h>
#include <globalPlatform_session/utils.h>

/* GP TEE Client function ID*/
#define INITIALIZE_CONTEXT 0
#define OPEN_SESSION 1
#define INVOKE_COMMAND 2
#define CLOSE_SESSION 3
#define FINALIZE_CONTEXT 4

/*print uint32_t and so on*/
#include <inttypes.h>

namespace GP {
   struct Session_component;
   struct Root_component;
   struct Main;
}

TEE_Param parameters[4];
```

```
int paramsUpdated = 0, getMemRefBuffer = 0, currentProcID = 0;
uint32_t currentSessID, currentSwID;
static process_list proc_list;

/* TODO create function that copies all uuids (in header) to variables to be used */
char *uuid_hello_ta = "8aaaf200-2450-11e4-abe20002a5d5c51b";

void setParams(char *param_a, char *param_b)
{
   uint32_t a, b;

   a = convertChar2uInt(param_a);
   b = convertChar2uInt(param_b);

   parameters[0].value.a = a;
   parameters[0].value.b = b;

   paramsUpdated = 1;
}

void setMemRefBuffer(char *buffer)
{
   parameters[0].memref.buffer = malloc(sizeof(buffer));
   Genode::memcpy(parameters[0].memref.buffer, buffer, Genode::strlen(buffer) + 1);

   paramsUpdated = 1;
   getMemRefBuffer = 1;
}

void retrieveParams()
{
  char *param_a, *param_b, *memref_buffer, *retrievedParams, *rest_buff;

   retrievedParams = getParameters();
   if(!getMemRefBuffer){
      param_a = strtok_r(retrievedParams, ",", &rest_buff);
      param_b = strtok_r(rest_buff, ",", &memref_buffer);
      setParams(param_a, param_b);
   }else{
      memref_buffer = strtok_r(retrievedParams, ",", &rest_buff);
      setMemRefBuffer(memref_buffer);
   }
}
```

```
int extractOperationID(char *input, char *next_args)
{
   // Extract Context ID from current request
   char *output, *rest_buffer;
   int op_id;

   output = strtok_r(input, ",", &rest_buffer);
   PINF("Operation ID: %s", output);
   op_id = atoi(output);

   if(rest_buffer != NULL){
      //Copy received data back to user level program
      snprintf(next_args, MAX_SIZE, rest_buffer);
   }

   return op_id;
}

char *extractUUID(char *input, char *next_args)
{
   // Extract UUID from current request
   char *output, *rest_buffer;

   output = strtok_r(input, ",", &rest_buffer);
   PINF("UUID ID: %s", output);
   if(rest_buffer != NULL){
      //Copy received data back to user level program
      snprintf(next_args, MAX_SIZE, rest_buffer);
   }

   return output;
}

int extractContextID(char *input, char *next_args)
{
   // Extract Context ID from current request
   char *output, *rest_buffer;
   int ctx_id;

   output = strtok_r(input, ",", &rest_buffer);
   PINF("Context ID: %s", output);
   ctx_id = atoi(output);
```

```
   if(rest_buffer != NULL){
      //Copy received data back to user level program
      snprintf(next_args, MAX_SIZE, rest_buffer);
   }

   return ctx_id;
}

char *extractSessionID(char *input, char *next_args)
{
   // Extract session ID
   char *output, *rest_buffer;

   output = strtok_r(input, ",", &rest_buffer);
   PINF("Session ID: %s", output);

   if(rest_buffer != NULL){
      //Copy received data back to user level program
      snprintf(next_args, MAX_SIZE, rest_buffer);
   }

   return output;
}

uint32_t extractFunctionCommandID(char *input, char *next_args)
{
   // Extract function command ID
   char *output, *rest_buffer;
   uint32_t commandID;

   output = strtok_r(input, ",", &rest_buffer);
   PINF("Command ID: %s", output);
   commandID = convertChar2uInt(output);

   if(rest_buffer != NULL){
      //Copy received data back to user level program
      snprintf(next_args, MAX_SIZE, rest_buffer);
   }

   return commandID;
}
```

```
uint32_t extractParamTypes(char *input, char *next_args)
{
   // Extract operation paramTypes
   char *output, *rest_buffer;
   uint32_t paramTypes;

   output = strtok_r(input, ",", &rest_buffer);
   PINF("Operation Parameters Type: %s", output);
   paramTypes = convertChar2uInt(output);

   if(rest_buffer != NULL){
      //Copy received data back to user level program
      snprintf(next_args, MAX_SIZE, rest_buffer);
   }

   return paramTypes;
}

void extractOperationParameterValues(char *input)
{
   char *op_param_a, *op_param_b, *rest_buffer;

   // Extract operation param valua a
   op_param_a = strtok_r(input, ",", &rest_buffer);
   PINF("Operation parameter value a: %s", op_param_a);

   // Extract operation param valua b and store rest in memref_buffer
   op_param_b = strtok_r(rest_buffer, ",", &rest_buffer);
   PINF("Operation parameter value b: %s", op_param_b);

   setParams(op_param_a, op_param_b);
}

void extractOperationParameterMemoryRefBuffer(char *input)
{
   char *memref_buffer = NULL, *rest_buffer = NULL;

   // Extract operation memory reference buffer content
   memref_buffer = strtok_r(input, ",", &rest_buffer);
   PINF("Operation memory reference buffer content: %s", memref_buffer);

   setMemRefBuffer(memref_buffer);
}
```

```
void setCurrentSessID(uint32_t sess_id)
{
   currentSessID = sess_id;
   setUserCurrentSessionID(sess_id);
}

uint32_t getCurrentSessID()
{
   return currentSessID;
}

void setCurrentProcID(int process_id)
{
   currentProcID = process_id;
   setUserCurrentProcessID(process_id);
}

int getCurrentProcID()
{
   return currentProcID;
}

void setCurrentProcList(){
   proc_list = getProcessLists();
}

struct GP::Session_component : Genode::Rpc_object<Session>
{
   int updateStatus()
      {
      return paramsUpdated;
      }

   void gp_api_handler(struct I2c_message *msg)
   {
      Genode::log("#### GlobalPlatform API Handler ####");

      // Parameter variables and containers
      char *pid, *op_id, *session_id, *uuid, *rest_buff, *rest_buff_process_id;
      int process_id, operation_id, context_id, i, j;
      char next_args[MAX_SIZE]= "";
```

```
  uint32_t commandID, paramTypes, sess_id;
  TEE_Result res;

  // Extract process ID
  pid = strtok_r(msg->bytes, ";", &rest_buff_process_id);
  process_id = atoi(pid);
  PINF("Process ID: %d", process_id);
  setCurrentProcID(process_id);

  // Extract Operation ID
  op_id = strtok_r(rest_buff_process_id, ".", &rest_buff);
  operation_id = atoi(op_id);
  PINF("Operation ID: %d", operation_id);

  switch(operation_id) {
     case INITIALIZE_CONTEXT :
       PINF("GENODE - INITIALIZE_CONTEXT\n");
         context_id = extractContextID(rest_buff, next_args);
         init_process_context(process_id, context_id);
         res = TA_CreateEntryPoint();
         if (res != TEE_SUCCESS){
              PINF("GENODE - Failed to create entry point\n");
         }
       break;
    case OPEN_SESSION :
    PINF("GENODE - OPEN_SESSION\n");

// Extract UUID and Session ID
 uuid = extractUUID(rest_buff, next_args);
  if(Genode::strcmp(uuid, uuid_hello_ta) == 0){

     context_id = extractContextID(next_args, rest_buff);
    session_id = extractSessionID(rest_buff, next_args);
     sess_id = convertChar2uInt(session_id);
     setCurrentSessID(sess_id);
     addSessionToList(process_id, context_id, currentSessID);
     paramTypes = extractParamTypes(next_args, rest_buff);
     setCurrentProcList();

     for(i=0; i <= MAX_PROC; i++){
      if(proc_list.processes[i].process_id == process_id){
        for(j=0; j <= MAX_SESSIONS; j++){
          if(proc_list.processes[i].nw_sessions[j].session_id == currentSessID){
```

```
          currentSwID = getSecureWorldSessionID(process_id, currentSessID);
          if(proc_list.processes[i].sw_sessions[j].session_id == currentSwID){
              res = TA_OpenSessionEntryPoint(paramTypes, NULL,
              proc_list.processes[i].sw_sessions[j].contextSession[j].session_ctx);
              if (res != TEE_SUCCESS){
                PINF("GENODE - Failed to open session entry point\n");
               }
              }
             }
          }
        }
    }
}
   break;
   case INVOKE_COMMAND :
       PINF("GENODE - INVOKE_COMMAND\n");

        session_id = extractSessionID(rest_buff, next_args);
        sess_id = convertChar2uInt(session_id);
        setCurrentSessID(sess_id);

        commandID = extractFunctionCommandID(next_args, rest_buff);
        paramTypes = extractParamTypes(rest_buff, next_args);

        if(commandID != 1){
          extractOperationParameterValues(next_args);
        }else{
          extractOperationParameterMemoryRefBuffer(next_args);
        }

        res = TA_InvokeCommandEntryPoint(NULL, commandID, paramTypes, parameters);

        if (res != TEE_SUCCESS){
        PINF("GENODE - Failed to invoke command entry point\n");
        }
        retrieveParams();
      break;
   case CLOSE_SESSION :
      PINF("GENODE - CLOSE_SESSION\n");

        session_id = extractSessionID(rest_buff, next_args);
        sess_id = convertChar2uInt(session_id);
        setCurrentSessID(sess_id);
```

```
            removeSessionToList(process_id, currentSessID);
            TA_CloseSessionEntryPoint(NULL);
            break;
        case FINALIZE_CONTEXT :
           PINF("GENODE - FINALIZE_CONTEXT\n");
            TA_DestroyEntryPoint();
            for(i=0; i <= MAX_PROC; i++){
               if(proc_list.processes[i].process_id == process_id){
                  for(j=0; j <= MAX_SESSIONS; j++){
                     if(proc_list.processes[i].nw_sessions[j].session_id ==
                         currentSessID){
                        currentSwID = getSecureWorldSessionID(process_id,
                            currentSessID);
                        if(proc_list.processes[i].sw_sessions[j].session_id ==
                            currentSwID){
                           free(proc_list.processes[i]
                           .sw_sessions[j].contextSession[j].buffer);
                        }
                     }
                  }
               }
            }
            break;
         default:
           PINF("[GlobalPlatform API Handler] - Operation Unknown\n");
         }
}

void getUpdatedData(struct I2c_message *ret)
{
   Genode::log("#### Getting updated data ####");

   char args[500] = "";
  char *cur = args, * const end = args + sizeof args;
  /* Copy all arguments needed by the the Secure World and send them through the
      smc */
  if (cur < end) {
      if(!getMemRefBuffer){
       cur += snprintf(cur, end-cur, "%l" PRIu32, (unsigned
           long)parameters[0].value.a);
       cur += snprintf(cur, end-cur, "%s", " ");
       cur += snprintf(cur, end-cur, "%l" PRIu32, (unsigned
           long)parameters[0].value.b);
```

```
        }else{
           /* Operation memory reference buffer */
           cur += snprintf(cur, end-cur, "%s", (char *) parameters[0].memref.buffer);
           getMemRefBuffer = 0;
        }
     }
      Genode::memcpy(ret->bytes, args, Genode::strlen(args) + 1);
      paramsUpdated = 0;
   }
};

class GP::Root_component
:
   public Genode::Root_component<Session_component>
{
   protected:

      Session_component *_create_session(const char *args)
      {
         Genode::log("creating GlobalPlatform session");
         return new (md_alloc()) Session_component();
      }

   public:

      Root_component(Genode::Entrypoint &ep,
                     Genode::Allocator &alloc)
      :
         Genode::Root_component<Session_component>(ep, alloc)
      {
         Genode::log("creating root component");
      }
};

struct GP::Main
{
   Genode::Env &env;

   /*
    * A sliced heap is used for allocating session objects - thereby we
    * can release objects separately.
    */
   Genode::Sliced_heap sliced_heap { env.ram(), env.rm() };
```

```
   GP::Root_component root { env.ep(), sliced_heap };

   Main(Genode::Env &env) : env(env)
   {
      /*
       * Create a RPC object capability for the root interface and
       * announce the service to our parent.
       */
      env.parent().announce(env.ep().manage(root));
   }
};

Genode::size_t Component::stack_size() { return 64*1024; }

void Component::construct(Genode::Env &env)
{
   static GP::Main main(env);
}
```

D GlobalPlatform TEE Internal API

tee_ta_api.cc

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Genode Includes */
#include <base/component.h>
#include <base/log.h>

/* TrustFrame + GlobalPlatform includes */
#include <globalPlatform_session/tee_ta_api.h>
#include <globalPlatform_session/user_defines.h>
#include <globalPlatform_session/user_defines_utils.h>
#include <globalPlatform_session/utils.h>

/*
 * Called when the instance of the TA is created. This is the first call in
 * the TA.
 */
TEE_Result TA_CreateEntryPoint(void)
{
   PINF("Called TA_CreateEntryPoint\n");
   return TEE_SUCCESS;
}

/*
 * Called when the instance of the TA is destroyed if the TA has not
 * crashed or panicked. This is the last call in the TA.
 */
void TA_DestroyEntryPoint(void)
{
   PINF("Called TA_DestroyEntryPoint\n");
   releaseCurrentContext();
}
```

```
/*
 * Called when a new session is opened to the TA. *sess_ctx can be updated
 * with a value to be able to identify this session in subsequent calls to the
 * TA.
 */
TEE_Result TA_OpenSessionEntryPoint(uint32_t param_types,
     TEE_Param params[4], void **sess_ctx)
{
   PINF("Called TA_OpenSessionEntryPoint\n");

   uint32_t exp_param_types = TEE_PARAM_TYPES(TEE_PARAM_TYPE_NONE,
                TEE_PARAM_TYPE_NONE,
                TEE_PARAM_TYPE_NONE,
                TEE_PARAM_TYPE_NONE);

   if (param_types != exp_param_types)
      return TEE_ERROR_BAD_PARAMETERS;

   /* Unused parameters */
   (void)&params;
   (void)&sess_ctx;

   /*
    * TEE Internal API doesn't
    * specify any means to logging from a TA.
    */
   PINF("[SW - Open Session] - Hello!\n");
   /* If return value != TEE_SUCCESS the session will not be created. */
   return TEE_SUCCESS;
}

/*
 * Called when a session is closed, sess_ctx hold the value that was
 * assigned by TA_OpenSessionEntryPoint().
 */
void TA_CloseSessionEntryPoint(void *sess_ctx)
{
   PINF("Called TA_CloseSessionEntryPoint\n");

   (void)&sess_ctx; /* Unused parameter */
   PINF("[SW - Close Session] - Goodbye!\n");
}
```

```
/*
 * Called when a TA is invoked. sess_ctx hold that value that was
 * assigned by TA_OpenSessionEntryPoint(). The rest of the paramters
 * comes from normal world.
 */
TEE_Result TA_InvokeCommandEntryPoint(void *sess_ctx, uint32_t cmd_id,
        uint32_t param_types, TEE_Param params[4])
{
   PINF("Called TA_InvokeCommandEntryPoint\n");

   (void)&sess_ctx; /* Unused parameter */

   uint32_t session_id = getUserCurrentSessionID();
   int process_id = getUserCurrentProcessID();

   selectFunction(cmd_id, process_id, session_id)(param_types, params);
   TEE_Result res = getResult();

   if(res != 0){
     return TEE_SUCCESS;
   }
   return TEE_ERROR_BAD_PARAMETERS;
}
```

E Hello World Trusted Services Example

user_defines.cc

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/* Genode Includes */
#include <base/component.h>
#include <base/log.h>

/* GlobalPlatform Include */
#include <globalPlatform_session/tee_ta_api.h>

/* Secure World Utils Includes */
#include <globalPlatform_session/user_defines.h>
#include <globalPlatform_session/user_defines_utils.h>

/*print uint32_t and so on*/
#include <inttypes.h>

TEE_Result incValue(uint32_t param_types, TEE_Param params[4])
{
    if(verifyParamTypesInout(param_types) != TEE_ERROR_BAD_PARAMETERS){
      setUserParameters(++params[0].value.a, params[0].value.b);
     return TEE_SUCCESS;
    }
    return TEE_ERROR_BAD_PARAMETERS;
}

TEE_Result copyStr(uint32_t param_types, TEE_Param params[4])
{
  if(verifyParamTypesMemRef(param_types) != TEE_ERROR_BAD_PARAMETERS){
    char received[MAX_SIZE] = "";
    char response[MAX_SIZE] = "";
```

```
    Genode::memcpy(received, (char *) params[0].memref.buffer,
              Genode::strlen((char *) params[0].memref.buffer));

    Genode::memcpy(response, "ACK", Genode::strlen("ACK") + 1);
    setUserMemoryRefBuffer(response);
    return TEE_SUCCESS;
   }
   return TEE_ERROR_BAD_PARAMETERS;
}

typedef struct data{
   uint32_t day, month, year;
} date;

TEE_Result printNextDate(uint32_t param_types, TEE_Param params[4])
{

  if(verifyParamTypesInout(param_types) != TEE_ERROR_BAD_PARAMETERS){
    date d;
    d.day = params[0].value.a + 1;
    d.month = params[0].value.b;
    /* No more than two value parameters defined in TEE_Param structure */
    d.year = 2016;
    setUserParameters(d.day, d.month);
    return TEE_SUCCESS;
   }
   return TEE_ERROR_BAD_PARAMETERS;
}

TEE_Result (*selectFunction(uint32_t cmd_id, int process_id, uint32_t
    session_id))(uint32_t param_types, TEE_Param params[4])
{
  TEE_Result (*funcSelect)(uint32_t param_types, TEE_Param params[4]);
  setUserCurrentProcessID(process_id);
  setUserCurrentSessionID(session_id);
  switch (cmd_id) {
  case TA_HELLO_WORLD_CMD_INC_VALUE:
    PINF("Calling command function to increment value\n");
    funcSelect = &incValue;
    setResult((TEE_Result) funcSelect);
    return funcSelect;
  case TA_FUNC_CPY_STR:
    PINF("Calling command function to copy a string\n");
```

```
    funcSelect = &copyStr;
    setResult((TEE_Result) funcSelect);
    return funcSelect;
  case TA_FUNC_PRINT_CMD_DATE:
    PINF("Calling command function to print next date\n");
    funcSelect = &printNextDate;
    setResult((TEE_Result) funcSelect);
    return funcSelect;
  default:
    return 0;
  }
}
```