Project acronym:	**TRESCCA**
Project title:	TRustworthy Embedded systems for Secure Cloud Computing
Project number:	European Commission – 318036
Call identifier:	FP7-ICT-2011.1.4
Start date of project:	01 Oct. 2012 **Duration:** 36 months

Document reference number:	**D3.2**
Document title:	Secure Hypervisor
Version:	1.3
Due date of document:	April 2015
Actual submission date:	May 2015
Lead beneficiary:	VOSYS
Participants:	Michele PAOLINO, Alvise RIGO, Daniel RAHO, Maria SOLER (VOSYS), Miltos GRAMMATIKAKIS (TEI), Renaud PACALET (TP)
Reviewers:	Bernhard Katzmarski (OFFIS)

Project co-funded by the European Commission within the 7th Framework Programme		
DISSEMINATION LEVEL		
PU	Public	X
PCA	Public with confidential annex	
CO	Confidential, only for members of the consortium (including Commission Services)	

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 Purpose of the Document

This document is the deliverable D3.2 of the TRESCCA project. Its aim is to describe the TRESCCA Secure Hypervisor, along with the software and hardware solutions developed to enhance the security of the virtualized system (both the guests and the host operating systems).

Chapter 2 details the concept of Trusted Compartment, a key component for the security of the TRESCCA platform. In fact, the Trusted Compartment is the secure container which includes the root of trust of the system. This area contains security assets and functions, and thus needs to be protected and well isolated from the security threats which may come from the user's applications. In order to provide a high security standard, the Trusted Compartment leverages the hardware capabilities of the TRESCCA platform to be more effective. Two different Trusted Compartment implementations are exploitable in the TRESCCA platform. The former is based on ARM TrustZone, a security framework designed by ARM and supported by most of the latest processors proposed by the company. The latter implementation of Trusted Compartment exploits the HSM-NoC developed in the TRESCCA project to provide isolation and protection of the TRESCCA platform's external memory. Both the above solutions are depicted in this document.

Similarly, Chapter 3 presents a description of the software stack used by the secure applications to request services to the Trusted Compartment. The Trusted Execution Environment (TEE), is a standardized solution designed to provide a flexible, device agnostic and portable security software stack. The standard specifications define a series of APIs which enables developers to easily create secure services (which will run in the Trusted Compartment) and secure application (programs which exploit the Trusted Compartment's services).

The implementation of KVM in the TRESCCA platform, its integration with TrustZone along with the virtualization of the TEE and its features are described in Chapter 4, while Chapter 5 details how the Secure Hypervisor will interact with the HSM modules.

All these technologies, combined together, form the TRESCCA Secure Hypervisor.

2 THE TRUSTED COMPARTMENT

This Chapter documents the concept of Trusted Compartment, its possible implementations (using either Trust-Zone or the HSM-NoC) and performance extensions through hardware acceleration.

The Trusted Compartment is very important for the TRESCCA platform, as it contains all the security related assets and functions of the system. As a matter of fact, it depicts the container of the TRESCCA system root of trust. System's keys, decrypt functions, algorithms and fingerprints are stored and executed in this area, which is for this reason the majority of the attacks' target.

In the next sections, two possible implementations of the Trusted Compartment will be discussed: the first based on ARM TrustZone, and the other based on HSM-NoC.

2.1 Trusted Compartment Implementation

Software based solutions to provide security and isolation of the modern electronic devices (from personal computers to mobile devices such as tablets and smartphones) have demonstrated limits in fending off attacks and providing assurance of current configuration [14]. For this reason, to protect the system while achieving high level of security and isolation, the implementation of the TRESCCA Trusted Compartment will be based on hardware components. These components will be detailed in the next sections, while the software stack used to interact with the Trusted Compartment will be described in Chapter 3.

2.1.1 TrustZone

The TrustZone architecture aims to provide a security framework that enables embedded systems protection [13]. Instead of providing a fixed security solution, such as what it is proposed by TPM (Trusted Platform Module) chips, it provides a flexible infrastructure that can be customized according to the needs of the particular SoC. In order to achieve flexibility, it offers a programmable environment that guarantees confidentiality and integrity against specific attacks.

In a TrustZone system, the applications' execution environment is split in Secure and Normal (or Non-Secure) World. The former is basically the Trusted Compartment, the place in which security assets are protected and confined while the Normal world is the user's system, where common operating systems such as Linux or Android are installed and programs/mobile Apps are run.

These two worlds are isolated by hardware signals, indeed ARM TrustZone is a system-wide approach to security, tightly integrated into Cortex-A processors and extended to the system's peripherals via the AMBA AXI bus and specific TrustZone System IP blocks. In fact, the AMBA AXI bus is aware of the security state of the application currently running in the processor, and ensures that secure world resources cannot be accessed from the Normal World. This approach allows to secure peripherals such as memory, crypto blocks, keyboard and screen in order to protect them from software attacks. ARM processors supporting TrustZone include v8 (ARM Cortex-A57, ARM Cortex-A53), and v7 (from the ARM1176 to the Cortex-A15) architectures.

Moreover, with TrustZone, a single physical processor core is able to execute code from both the Normal and the Secure worlds, thus eliminating the need for a dedicated security core. A single core is thus split in two virtual processors, one corresponding to the Normal domain and another for the Secure domain. The context switch between these domains is performed via the monitor mode, resulting in a change of the running virtual

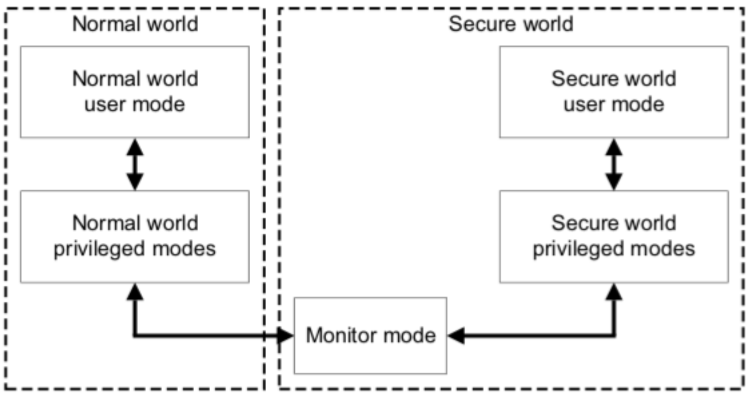


Figure 2.1: ARM TrustZone Secure and Non-Secure Worlds (source:ARM Security Technology [13]).

processor. The mechanisms to enter into the monitor mode from the Normal world are tightly controlled and include the Secure Monitor Call (SMC) instruction and a subset of the hardware exception mechanisms. The software that executes in monitor mode is implementation defined, but it generally behaves as a context switch manager between the two different domains.

In order to fully exploit the ARM TrustZone capabilities, there is need of software components in both user and kernel space. To prevent the proliferation of specific and custom solutions, and to foster the growth of a secure application developers community, ARM decided to support the GlobalPlatform Trusted Execution Environment (TEE) specifications. More information about the GlobalPlatform can be found in Chapter 3.

Finally, the TrustZone hardware architecture also includes a security-aware debug infrastructure that can enable secure world debug, without impairing debug visibility of the Normal world.

2.1.2 HSM-NoC

The HSM-NoC is a novel hardware IP component providing isolation in integrated System-on-Chip solutions. It is always placed between a Network-on-Chip and a system component (e.g. memory controller or hardware accelerator), ideally at the device's network interface. The HSM-NoC acts as a firewall between the NoC and the system component, adapting concepts from enterprise network firewalls to the level of the on-chip communication architecture of multicore SoC devices. Thus, a NoC Firewall is able to resist malevolent or malformed NoC communication, detect vulnerabilities that arise from NoC protocol violations perpetrated by erroneous or malicious IPs, and improve network performance and dynamic power. Typically there are multiple HSM-NoCs distributed in the system, i.e. one HSM-NoC for each IP connected to the Network-on-Chip. Within each HSM-NoC component different access rules can be defined, thus regulating access to the memory regions protected by the HSM-NoC.

This means that each process (and thus each virtual machine) can set its own rules for the RAM, protecting a portion of the system memory to implement the Trusted Compartment.

2.2 Security HW accelerators

Hash, cryptographic and other security algorithms (e.g. SHA, AES, MD5, RSA, etc.) are computationally-intensive and require high CPU power and energy. For this reason, in embedded and mobile systems, HW acceleration is used to achieve better performance and save power. Typically the implementation of these algorithms was performed directly in hardware, providing very good performance but with a limited flexibility.

In the last years, many scientific works focused on the implementation of security algorithms using programmable HW accelerators such as GPGPU [16] and FPGA [6].

The TRESCCA Trusted Compartment supports security hardware accelerators to enhance the performance of security functions while keeping low power consumption. In this way, the secure application running in the Trusted Compartment are able to offload CPU-intensive tasks to standard and programmable HW accelerators. This provides to the TRESCCA application high performance combined with low power consumption and the possibility to update or change dynamically the security algorithm used in the Trusted Compartment.

The Trusted Compartment owns and controls the HW accelerators, but is also able to share them with the Non-Trusted Compartment when computational power is needed for virtual machines and their applications.

3 THE TRUSTED EXECUTION ENVIRONMENT (TEE)

The present chapter is organized in three sections, first an overview of the Trusted Execution Environment (TEE) with some introductory lines is presented (Section 3.1), followed by a description of the GlobalPlatform Trusted Execution Environment (TEE) standard (Section 3.2), including its benefits for the TRESCCA platform. Soon after, a study of the current TEE state of the art is presented (Section 3.3)

3.1 Introduction to the Trusted Execution Environment(TEE)

The TEE is the GlobalPlatform name for a secure area that resides in the main processor of an embedded device (such as a smartphone or a tablet) and ensures that highly sensitive data is stored, processed and protected in a trusted environment [24]. Thus in the TRESCCA environment, as in the rest of this document,TEE and Trusted Compartment are synonymous. Historically, Trusted Execution Environments were completely isolated from open (non Trusted) environments, meaning that they could not coexist in the same device, but they had to live in separate devices. The concept of Trusted Execution Environment, allows to break this barrier by proposing different means of isolation. The TEE grants the safe execution of authorized security software, known as 'trusted applications', providing end-to-end security by enforcing protection, confidentiality, integrity and data access rights. This is obtained through a combination of hardware and software isolation. However the hardware isolation does not imply the use of other devices, but of security enabled devices (such as ARM processors that include TrustZone).

As has been seen, the idea of TEE implies the existence of two separate domains of execution within a single device. In GlobalPlatform, these worlds are denominated Trusted Execution Environment (TEE) and Rich Execution Environment (REE). The TEE is a secure section that can be leveraged to store and execute sensitive data and operations. It runs along with the REE, where a Rich Operating System (Rich OS) offers versatility and a rich variety of applications, such as Android for example. The REE is open to download and installation of any software available and compatible with the device. This results in security breaches which make this kind of environment inappropriate to manage sensitive data or applications due to its vulnerability to malware or malfunction. The TEE also provides security services to the REE, increasing the security capabilities of the REE. The REE is, in the ARM TrustZone terminology, the Non-Secure World.

The concept of a TEE was originated due to the extensive use of mobile technologies in the last years. With this generalization of mobile technologies, services have emerged that require a greater level of security. In the same way, with an increasing number of users comes a greater need for protection from malware/viruses, specially for applications with higher security requirements. Corporate environments and the rise of mobile financial services require increased levels of security, including content protection and safe execution, to preserve confidentiality and integrity. Applications with higher security requirements, require more protection than can be offered by software solutions alone. The TEE isolates secure applications, keeping them away from any potential malware residing in the REE.

In the past device manufacturers have developed trusted environments and included them in their devices as a part of their proprietary solution. There was however no well defined API, no standard, so applications needed to be developed differently for each individual proprietary solution, limiting their portability and interoperability. Security sensitive service providers do not want to develop different versions of the same application for each proprietary TEE environments. A considerable effort is required to certify secure software; also, when creating several versions of the same software the potential error rate increases noticeably. It is thus important to reduce diversification of the software by means of standards such as GlobalPlatform TEE.

It is a device and Rich OS agnostic standard, that can be embraced by all suppliers and reside comfortably alongside any rich OS environment. The standardization of the TEE will overcome compatibility problems and will offer a uniform API that will ease secure application development, while supplying the necessary security.

3.2 The GlobalPlatform Standard

Trusted environments have been present for some time in the market, but they were proprietary dependant and there was no standard solution. As already mentioned in the TRESCCA Deliverable D2.1, the GlobalPlatform has developed the TEE Standard as a means to cope with the need for uniformity between trusted environments. GlobalPlatform is a non-profit association integrated by several industries, which develops and publishes specifications for secure chip technology. This association aims to conciliate security and interoperability. The main contributions of GlobalPlatform are the concepts of Secure Element (SE) and Trusted Execution Environment (TEE). As conceived by the GlobalPlatform, the TEE shares the device with the REE, with a holistic solution that integrates both a hardware and a software architecture, without dictating a particular implementation of either, but offering the security principles and software APIs to build one. This solution is still under development and will be extended in consecutive phases. In this section we focus in the software part of the GlobalPlatform Standard.

The software of the GlobalPlatform TEE is basically a set of APIs that enables communication between the REE and the TEE, but allows also the Trusted Applications to communicate with one another and with the Trusted Operating System (Trusted OS). Figure 3.1 shows the TEE Software Architecture, including the relationship between the major components.

Three are the main actors of the TEE software infrastructure: those that enable communication with the REE, the Trusted OS and the components that allow communication of the Trusted Applications.

As for the TEE-REE communication, the currently defined elements are the TEE Client API [22], and the TEE Communication Agent. The TEE Client API is an interface that allows Client Applications in the REE to communicate with Trusted Applications in the TEE, while the TEE Communication Agent provides support for messaging between the Client Application and the TEE. There is another element which is supposed to run in the TEE, the Functional API. Since it has not yet been fully specified in the GlobalPlatform specification, this component will not be further discussed in this document.

The Trusted OS Components are the Trusted Core Framework, that enables OS like functionality for Trusted Applications, the Trusted Functions, that provide facilities for development, the TEE Communication Agent, counterpart of the REE Communication Agent in the REE, and finally the Trusted Kernel.

Regarding the Trusted Applications (TA), their communication is performed through the TEE Internal API [23], that defines the fundamental software capabilities of a TEE. Other APIs may be defined to this effect.

It is also important to mention that the GlobalPlatform TEE specification is built on top of the system

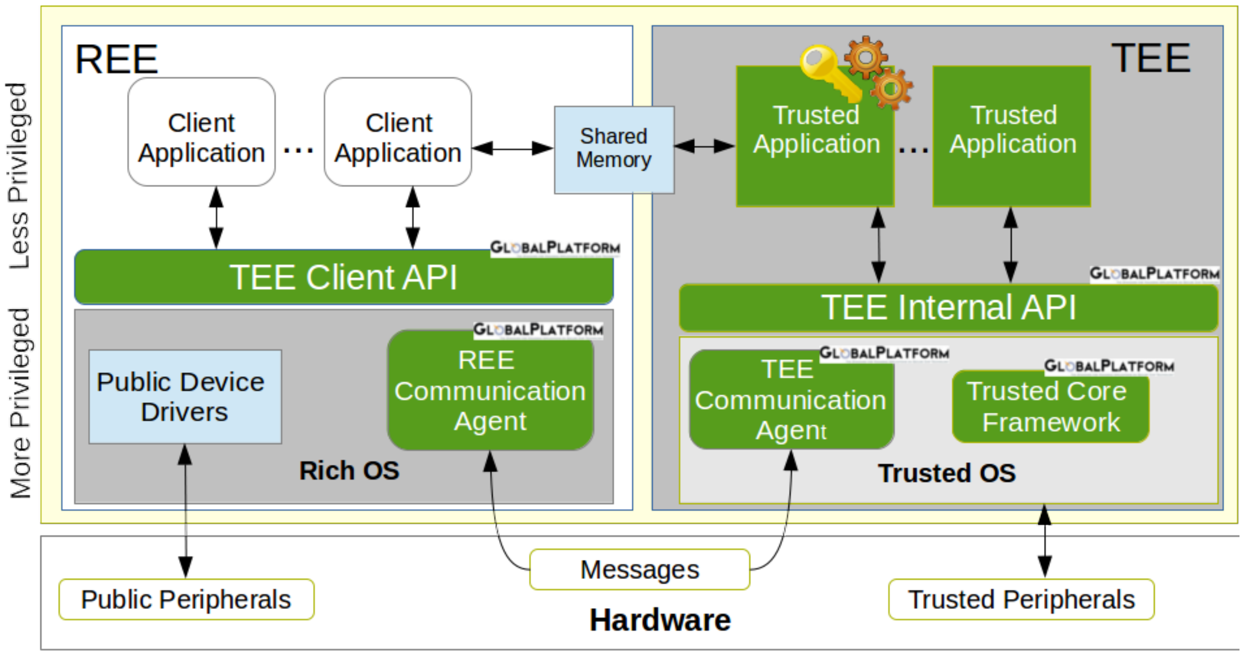


Figure 3.1: GlobalPlatform TEE Software Architecture.

"chain of trust", which is what provides reliability to the TEE. The root of this chain is called root of trust, and it is based on the hardware features used to protect cryptographic keys, perform authentication and verify software. Software wise, the root of such a chain is the booting process (more concretely the bootloader), since this is the first step taken during the execution. The bootloader verifies the authenticity of the Trusted OS and boots it. Once booted, the Trusted OS checks and starts the bootloader of the REE, which verifies the authenticity of the Rich OS and eventually boots it. If the root of trust is not breached and these steps are followed, then the system is dependable. Of course the REE side is not to be trusted due to the excessive variability it introduces, but some parameters can be still checked and controlled from the TEE side.

After this introductory explanation of the overall GlobalPlatform TEE Software Architecture, the TEE Client API and the TEE Internal API will be explained in detail.

3.2.1 The GlobalPlatform TEE API

As has been seen in the previous section, the GlobalPlatform TEE API is mainly composed of three components, the TEE Client API, an interface that allows communication between the TEE and the REE (Figure 3.1); the Trusted Core Framework (TCF), that provides OS like functionality to the TEE and the TEE Internal API, which allows the communication between the TCF and the Trusted Applications, as well as the communication between Trusted Applications. In this section a more detailed explanation of each of these components is offered. The complete specification of the TEE Client API can be found in [22]. Both the Trusted Core Framework and the TEE Internal API are defined in [23].

A typical TEE API command flow starts with the CA asking a Security Context for all its calls to that TEE. Once the Context is initialized, it can open a Session with any of the Trusted Applications (TAs) within that TEE. The same CA can open several sessions with the same or different TAs. Within each session, a CA can invoke any of the Services (commands) offered by the TA. Once the interaction between the CA and the

TA is finished, the CA can close the Session, and once all Sessions have been closed and the TA has finished interacting with the TEE, it can finalize the Context.

Before going further in the explanation of the components of the GlobalPlatform TEE API, it is important to note that when we talk about Client Applications, it can refer to REE applications or to Trusted Applications that make use of the services of another Trusted Application. We will see that the mechanisms of communication are different in each but it is transparent to the Trusted Application and it is the TCF that will take care of these differences.

The Client API

The TEE Client API is an Inter Process Communication (IPC) API that deals with the communication between the REE and the TEE. This allows the applications in the REE (Client Applications or CAs) to leverage the services offered by the TEE, such as cryptography or any other. This API has been created in a most generic way, to allow for communication without restricting the services the TEE can offer to the REE.

The Trusted Core Framework API

The Trusted Core Framework API is a common abstraction layer on top of which all the Trusted Application are executed. It includes:

- The *TA Interface*: it represents the API between the TCF and the Trusted Applications. The TCF calls these functions in the Trusted Application, to load the application, create Sessions between the Trusted Application and a Client Application, invoke the commands of the TA, close the Session and terminate the application. This means that the TCF has complete control over the TA and that all communications between Client Applications and Trusted Applications are made through the TCF.
- The *property access functions*: they allow to access properties in a property set. These functions can be used to access TA Configuration Properties, Client Properties, and Implementation Properties (Trusted OS/TCF Properties).
- The *Panic function*: It allows the TEE to stop a TA completely should an exception happen. Panic causes are clearly defined in the [23] as well as the effects of the Panic function. The TA must be gracefully closed avoiding any possible further failure of the system.
- The *Internal Client API*: It allows for the communication among TAs. When the Client Application is a TA it uses the Internal Client API to obtain services from another TA.
- The *Cancellation function*: It allows to cancel a service request which has not yet been started. If it has been already started, the task is set in canceled state. It has effect only in few tasks and it can always be masked.
- The *Memory Management Functions*: They allow to allocate, free, copy, fill and manage the memory of the TAs in a way similar to POSIX. A function to check the access rights of a given buffer as well as one to access an instance data register are also included.

The TEE Internal API

The TEE Internal API includes the Trusted Storage for Data and Keys, the TEE Cryptographic Operations, the Time, and the TEE Arithmetical APIs. These allow the TAs to leverage the services offered by the TEE. The *Trusted Storage API* allows for the storage of both data and keys in a secure way; the *TEE Cryptographic Operations API* allows the TAs to use an encryption/decryption library with standard functions; the *Time API* offers several time sources (both secure and non-secure) and a wait function; finally the *TEE Arithmetical API* provides Big Integer functions as a solution to add cryptographic functions not included in the TEE Cryptographic Operations.

3.3 The TEE state of the art

As has been seen in the previous sections, Security in ARM platforms has become an important target of research and development in the late years. GlobalPlatform's Trusted Execution Environment is a standard that several companies have followed. There are thus several solutions both open and closed source that claim to be complete. In the following section, an overview of these component is presented.

3.3.1 Closed Solutions

Among the existing, closed TEE solutions, the one provided by Sierraware is the most documented. The information related to the other solutions mainly comes from marketing and advertising messages.

SierraTEE

SierraTEE is a secure operating system developed for ARM TrustZone hardware security extensions, combined with an implementation of the GlobalPlatform TEE to allow communication of the two running OSes: one executed in the Non Secure World and another in the Secure World. SierraTEE covers a wide range of ARM architectures like ARM11, Cortex-A8, Cortex-A9 and Cortex-A15. It is combined with SierraVisor Hypervisor, which enables the paravirtualization of several ARM processors, and hardware (full) virtualization of the Cortex-A15. Since this solution is completely closed, it is not possible to assess whether they are providing Secure World access to the VMs through a Virtualized TEE Client API. SierraWare is available for HiSilicon Multimedia Processors (used for Digital Right Management or DRM) as well as Zync-7000 AP SoC that targets a wider market including aerospace and defense, automotive, broadcast consumer, high performance computing, industrial, scientific, medical and wired and wireless communications.

SierraWare also has an open source solution, OpenVirtualization, that will be described later in this document.

Other closed TEE solutions

The technical documentation related to these solutions is scarce or not present. For this reason the following overview of the most important closed solution may be incomplete or not updated:

- **Texas' Instruments M-Shield TEE**: Android 4.0 on OMAP 4 uses M-shield TEE for DRM purposes. It is also included in all OMAP processors, which are mainly used in DRM and automotive infotainment.
- **Inside Secure**: It is used mainly for DRM applications. For content protection Inside's Fusion products offer HDCP and DTCP-IP solutions to secure High Definition (HD) video content for wired and wireless device-to-device streaming; it can also be used in embedded DRM Fusion agents to support Microsoft PlayReady. For enterprise security, it protects the users information and secure communications over IPSEC and SSL through its SafeZoneFIPS- Certified Cryptographic modules, and QuickSec and Matrix software development kits.

Solution	SierraTEE	M-Shield TEE	InsideSecure	Trustonic
Support Cortex-A15	Y	Y	N	Y
Support virtualization	Y	N	N	N
Certification	N	N	N	Y
Available since	2011	2013	2013/2014	2013
ARMv8	Y	No mention	No mention	Y

Table 3.1: Comparison of the main aspects of the Closed Source TEE solutions.

- **Trustonic**: This solution is certified. The product is called TEE Directory and is used to allow trusted service managers to offer protected services. Trustonic technology is embedded in over 100 smart connected devices. Whilst there is no information on the companies using the TEE Directory, there is a number of partners (over 40) that participate. These partners range from chip and device makers to security developers, including also commerce, MM content, enterprise, HW systems, identity management and network operators.

Closed TEE summary

Table 3.1 summarizes the main characteristics of the closed solutions. Its content depicts how TEE vendors are working to provide solutions with virtualization support. It's interesting to notice that SierraTEE is compatible with the next generation of ARM processors (ARM v8). Both these features are targeted by the TRESCCA TEE. On the other hand, it is also clear that only Trustonic provides a certified solution.

3.3.2 Open Source Solutions

There are three main Open Source solutions publicly available. From the three, only the code of OpenVirtualization is downloadable at the writing time. In this section, these three solutions will be detailed and discussed.

Open Virtualization

Being Open Virtualization based on the SierraWare closed solution, it includes an open version of SierraVisor and SierraTEE, a Trusted Execution Environment for ARM TrustZone hardware security extensions. It supports ARM11, Cortex-A9, and Cortex-A15 processors.

The OpenVirtualization is mostly a development environment, since it lacks important components (such as the bootloader) to be a full product. Indeed, its code is incomplete and some important features are missing (such as user space task isolation, kernel and user space separation, multitasking, dynamic application loading, secure boot and a POSIX compliant libc). It is also poorly documented. It is however the most complete Open Source version of the TEE whose code is actually available and the only one that claims to allow virtualization.

OpenVirtualization's components include:

- SierraVisor
- Secure World Kernel
- TEE Internal API: this component is largely incomplete.
- Trustzone API interface driver: A Kernel module for the normal world operating system, responsible for communicating with the secure world.

Solution	OpenVirtualization	T6	ANDIX OS
Support virtualization	Y	In progress	N
Crypto library used	wrap of OpenSSL	wrap of PolarSSL	tropicSSL
Secure OS	Custom	Custom	Custom
Dev. environment available	Y	N	N
Secure boot	N	In progress	Y
Available since	2013	March 2014	April 2014

Table 3.2: Comparison of the main aspects of the Open Source TEE solutions.

- TEE Client API

T6

T6 is an open source operating system for Trusted Execution Environment(TEE) using ARM's TrustZone. T6 is incomplete, it is intended for both the research community, and to be deployed in real mobile devices. Its kernel is based on XV6 [27]. However, there is not much information about the project and the code cannot be downloaded from the provided git repository.

ANDIX OS

ANDIX OS is as a multitasking, non-preemptive operating system for the TrustZone Secure World, but also includes all the necessary components to be run in the Non Secure World. The author presents the project as open source, but the source code is not yet available on the official website.

Open TEE summary

Table 3.2 is a summary of the Open Source solutions. It is interesting to notice that none of the presented solutions is certified or ready to be used in an ARM platform. Some of them support virtualization and provide a development environment to create new Trusted Applications. In all the cases, the secure OS chosen to power the Trusted Compartment is a custom solution, with sources not available in most of the cases.

4 THE KVM TEE

KVM (Kernel-based Virtual Machine) is the reference hypervisor for the TRESCCA project. It supports many CPU architectures (ARM, x86, PowerPC, S/390, etc.) and is completely integrated in the Linux kernel [10].

The TRESCCA platform uses virtual machines to isolate the applications and offer high standard of security and flexibility. In this section, the security of the Virtual Machines and the guest applications will be documented along with the implementation of the TRESCCA Secure Hypervisor.

Moreover, in regard to the topics discussed in this section, Virtual Open Systems (VOSYS) filed a patent application in March 2015.

4.1 The KVM hypervisor

The Linux Kernel Virtual Machine (KVM) is considered the most popular hypervisor deployed in OpenStack [3]. It is implemented as a Linux kernel module, which effectively turns the Linux kernel into a hypervisor, explotitng the ARM Virtualization Extension to create a fully-featured virtualization environment providing hardware isolation for CPU, memory, interrupts and timers [4]. The ARM Virtualization Extensions, in particular, allow certain instructions to trap to the hypervisor, include functionality to assist with the guests memory virtualization[1] and introduce a new processor mode (i.e., the hypervisor mode) which allows each guest to have access to its own privileged process execution mode.

The KVM approach takes advantage of the existing Linux kernel infrastructure, including the scheduler and memory management. This results in a tiny code base, compared to other hypervisors such as XEN or VMWare ESX. KVM works by exposing a simple `ioctl` interface, through which a regular Linux process can request to be turned into a virtual machine. When using QEMU/KVM, the QEMU [2] emulator is this process, which leverages on KVM for the precessor and the memory virtualization and provides an extensive set of software device implementations used to emulate I/O guest devices. Thus, KVM will handle CPU context switching when the process of a virtual machine gets scheduled by the Linux kernel, while the QEMU emulator takes care of the VM's I/O capabilities (e.g., storage, networking, shared memory, etc.) of each guest.

4.2 TEE Implementation and virtualization

For the development of the TEE for the TRESCCA platform, VOSYS used ARM TrustZone. Possible alternatives were HSM-Noc and TPMs (Trusted Platform Module) which has been leveraged in other projects [7]. TrustZone has been chosen for TRESCCA because it is the only solution which provides a software model (ARM FastModels) and a development board (ARM Versatile Express) with Virtualization Extensions. Moreover, TrustZone is a technology with increasing presence in the embedded market due to its flexibility as compared to TPM which are fixed-function devices with a predefined hardwired feature set.

In the TRESCCA implementation based on ARM TrustZone, the REE will be implemented inside the ARM Non-Secure World, while the TEE will be implemented in the ARM Secure World.

The TRESCCA Secure Hypervisor developed by VOSYS supports virtualization and targets ARM v7 and

[1]ARM Large Physical Address Extension is used for this. It provides a page table format which enables the possibility to set up a second stage of memory translation to be used by the hypervisor.

	OPTEE OS (0.2.0)	**FIASCO.OC/L4Re (snapshot 2014092821)**	**Linux kernel (4.0)**
source files (.c)	82	2725	14346
header files (.h)	35	2085	3287
total	117	4810	17633

Table 4.1: Code base estimation of OPTEE, FIASCO.OC/L4Re microkernel and Linux kernel (thousands of lines).

v8 architectures. The former empowers all the ARM based smart device currently on the market while the latter, more powerful and power efficient, is targeting the server market and the next generation mobile devices.

4.2.1 Secure World

One of the most important parts of the TRESCCA Secure Hypervisor is the software running in the TrustZone Secure World. The operating system running in the Secure World should be fast, secure, ideally real-time and free of programming errors (implementation correctness). In the next sections, two different software approaches for the population of the Secure World are described: A powerful, extensible and real-time micro-kernel (FIASCO.OC), and a thin library (OPTEE OS).

The primary motivation behind microkernels is the small code footprint if compared with standard OSes, which lead to a smaller attack surface, an easier process of formal verification of the code and usually a better performance. On the other hand, as shown in Table 4.1 [2] , the library approach (i.e. OPTEE OS) provides an even smaller code base. However, such a solution is less flexible and extensible than a microkernel. In the context of the TRESCCA project, both solution have been explored and evaluated, aiming to provide the best solution for each scenario.

Secure World as a microkernel: FIASCO.OC

The execution of a complete operating system in the ARM TrustZone secure World enables the TRESCCA platform to execute different types of secure applications. The chosen secure OS for the powerful ARMv8 processors is the Fiasco.OC Real Time microkernel, with its thin user space layer L4Re (Figure 4.1).

Fiasco.OC is a real time (and open source) microkernel 4.1 that runs on several platforms, including ARM. The microkernel is the lowest layer of software running in an L4 system. It provides primitives to execute tasks, to isolate them, and for safe communication between them. Fiasco.OC kernel services are implemented in kernel objects. Tasks hold references to such kernel objects in their "object spaces", kept as a kernel-protected table. These references are known as capabilities. Fiasco system calls are invocations of kernel objects through the corresponding capabilities. The kernel is the most privileged, security-critical software component in the system, so Fiasco.OC has been designed as light as possible in order to reduce its attack surface. It thus provides only the very minimal support for applications, any extra facility (such as device drivers) is provided by the L4 user space layer.

On the other hand, the L4Re userspace will run an implementation of the GlobalPlatform Internal API, the secure device drivers and the TAs. In order to do this, the Secure World OS does not use the main platform storage device to store files and the security assets of the system. An external, non-volatile memory configured by the Secure World as not accessible by the Non Secure World, is used to this purpose.

The L4Re layer provides a basic set of services and abstractions, that allow the implementation and execution of user-level applications. A minimal L4Re-based system consists of three basic components:

[2]These results consider the length (in lines) of all the .c and .h files, including comments. They are obtained executing the **bash** command `find . -name "*.h" -print | xargs wc -l | grep total`.

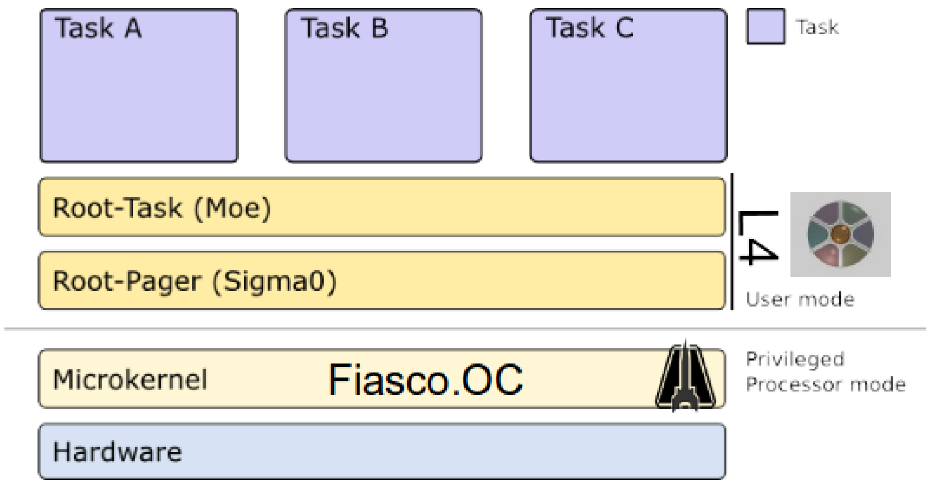


Figure 4.1: Fiasco-L4Re based system.

Fiasco.OC, the root pager (Sigma0), and the root task (Moe). Sigma0 initially owns all system resources, but is usually used only to resolve page faults. Moe provides essential services to user applications such as a program loader or virtual memory management.

The TRESCCA L4Re TEE

The TRESCCA L4Re TEE, developed by VOSYS, includes:

- Fiasco-L4Re as Secure OS
- L4Re TEE Internal API
- VOSYS shared memory [19], a QEMU/KVM device which enables the communication between TEE and REE.
- Linux TEE Client API
- Updated version of the L4 OpenSSL library (HeartBleed free)

A demo of the TRESCCA TEE based on L4Re has been shown during the second TRESCCA review meeting held in Bruxelles, on October 28th 2014. The demonstrator showed a REE and a Trusted Compartment implemented as two QEMU VMs, one running Android while the other Fiasco-L4Re(Figure 4.2). The two guests communicate through the TEE Client API, and use the TEE Internal API and the TCF on Fiasco-L4Re to provide security services. In the demo, the Android VM runs a DRM Music player application, which is able to play encrypted files using the security services of the TEE (the key and the algorithm are stored in the Secure World). Moreover, the TRESCCA L4Re relies on OpenSSL as cryptographic library: VOSYS has upgraded the existing port of OpenSSL for L4 [11] to a more recent version which resolves the well-known HeartBleed bug [8]. Lastly, the communication between VMs has been implemented through vosyshmem [19], a QEMU device developed by VOSYS to enable a full duplex communication between the two VMs.

Secure World as a thin library: OPTEE OS

OPTEE OS (also known as OPTEE only) represents a different approach for the implementation of the TEE. Despite its name, OPTEE OS is not a full fledged and complete operating system, but it is a set of libraries which implement the GlobalPlatform TEE Internal and Client API.

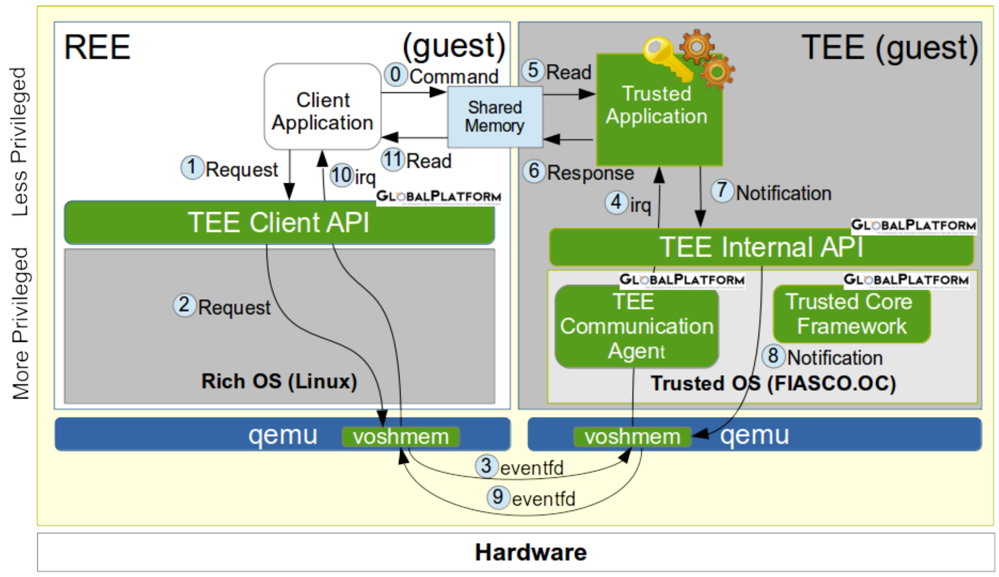


Figure 4.2: TRESCCA L4Re TEE demonstrator.

Being just a thin library, the OPTEE OS code base is smaller than the microkernel solution presented above as shown in Table 4.1. This project targets only ARM cores with TrustZone and adheres to GlobalPlatform TEE Client API 1.0 and TEE Internal API 1.0 specifications.

OPTEE OS started as a ST/Ericsson proprietary project, which has then been released as open source code in collaboration with Linaro [1]. Currently, the project consists of:

- The TEE Client API [3] released under a custom license, which includes tee-supplicant, a REE daemon which provides miscellaneous features such as file system access.
- The Linux driver [4] for the REE, which has been released under GPLv2 licence.
- The TEE Internal API [5], released under BSD licence. This specific licence enables SoC vendors and device manufacturers to implement their own OPTEE OS version without any obligation to disclose the modifications.

The TRESCCA OPTEE TEE

OPTEE OS can be deployed on a TrustZone enabled platform (such as the ARM Versatile Express) or, as an alternative, can be be run on a KVM virtual machine. This last solution, which emulates the Secure Word using QEMU, is recommended for development purposes only. During DATE 2015 conference (March 9-13th, Grenoble, France), VOSYS has shown a demo in which a single VM emulating TrustZone was executing a secure application leveraging on the OPTEE OS software infrastructure. In this demo, the application running in the REE is a DRM ebook reader, which sends the contents of the encrypted files to the TEE to decrypt them. After the decryption, the TEE sends the ebook data back in the REE, which shows the content of the ebook, without saving it in the local storage.

[3] TEE Client API code is available at https://github.com/OP-TEE/optee_client
[4] OPTEE Linux driver code is available at https://github.com/OP-TEE/optee_linuxdriver
[5] OPTEE TEE Internal API and OPTEE OS code is available at https://github.com/OP-TEE/optee_os

4.2.2 TEE virtualization

The virtualization of the TEE is of utmost importance for the TRESCCA Secure Hypervisor, because it links together the applications run in the VMs with the secure services available in the TrustZone Secure World. This process is completely transparent from the type of secure world implementation seen in the previous sections, as the GlobalPlatform APIs acts as an abstraction layer between the VMs and the TEE. At the time of writing, the guest OSes TEE access is not allowed in the GlobalPlatform API Specification. To enable this feature, the Secure Hypervisor virtualizes the GlobalPlatform TEE APIs, executing the TEE Client API directly in the Guest OS.

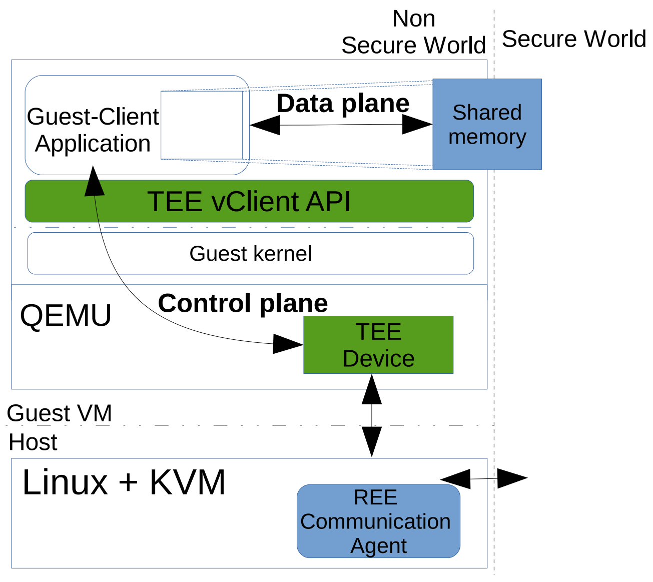


Figure 4.3: TEE support for Virtual Machines in TRESCCA.

In order to be as much as possible compliant with the GlobalPlatform Specification and to be able to run Client Applications (CAs) also at the host level, the TRESCCA TEE Client API is the only virtualization aware component.

This awareness needs support from the hypervisor infrastructure, for this reason, as depicted in Figure 4.3, a specific QEMU device is used to implement the TEE control plane and set up its data plane. All the requests (e.g., initialization/close session, invoke command, etc.) and notification of response are sent to the TEE Device, which delivers them either to the TAs or to the CAs running in the guest OS. To provide good data throughput and latency performance, the data plane is based on a shared memory mechanism [19]. Thus when a notification response arrives from the TrustZone Secure World, the TEE device notifies its driver with an interrupt, which forwards the related information to the Guest-Client Application. The Guest-CA is now able to read the data from the shared memory, without involving the TEE device in the data transfer.

4.3 Features

The KVM TEE Features are the basic mechanisms or functions that enable TEE Services to offer complete security solutions. These are: VMs Isolated execution and Trusted Communication.

4.3.1 Isolated Execution

The TRESCCA Secure Hypervisor relies on four different isolation layers, which are combined together to offer security and data protection for VMs. The first two, TrustZone and GlobalPlatform TEE API are detailed respectively in Section 2.1.1 and Section 3.2.

The others, KVM and SELnux/svirt, will be described in the following sections.

KVM hypervisor isolation

KVM is the open source hypervisor included in the Linux kernel. Despite its main goal is to run different operating systems concurrently, it offers interesting isolation capabilities. As a matter of facts, KVM creates virtual instances of the CPUs, memory and interrupts to provide an illusion of a real machine in software, which is used to run unmodified operating systems in guests. In order to achieve this, KVM on ARM provides hardware isolation for these resources exploiting hardware features such as Virtualization Extensions, IOMMU and GIC. Other resources such as device peripherals (network, disks, etc.) or shared memory are virtualized through emulation/para-virtualization and isolated in software, mainly using Discretionary Access Control (DAC) or Mandatory Access Control (MAC) implementations such as SELinux.

This type of isolation, although with some differences in the implemenation and efficiency, comes for free when using hardware assisted virtual machines with any hypervisor and CPU architecture.

SELinux/svirt

SELinux implements the MAC security policy in the Linux kernel[6] , which is used on top of the DAC, the security policy used by default. DAC is simple and easy to use for common users, but has important security drawbacks because it allows the owner of a resource (e.g., files, sockets, etc.) to delegate rights over it and supports only two role types, normal user and super user (also known as root). The latter, in particular, could become a security threat, as it has full control of the system. These limitations are an important concern in virtualized systems, where guests which belong to different entity could be hosted in the system of a service provider. In this scenario, using DAC only, the system administrator is able to tamper with the resources of the VMs (e.g., disks, shared memory, sockets, etc.). Moreover, a malicious guest has potentially access to all the files and resources of the other VMs and the host itself.

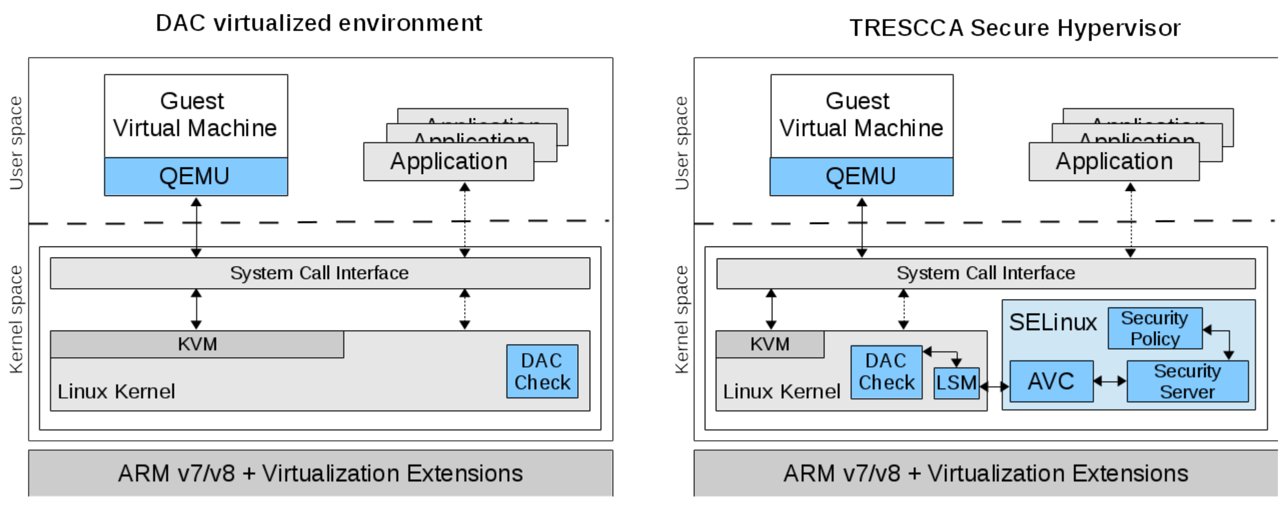


Figure 4.4: Comparison between DAC and SELinux TRESCCA Secure Hypervisor.

To solve these problems and enhance system security, the TRESCCA Secure Hypervisor uses SELinux,

[6]Alternative implementations of the MAC security policy in the Linux kernel are: TOMOYO, AppArmor and SMACK. For the TRESCCA Secure Hypervisor, SELinux has been chosen because it is considered the most mature and widely deployed amongst the Linux security mechanisms [21].

which has been developed at the early stages as a National Security Agency (NSA) proprietary project, but it is now fully open source and merged in the Linux kernel code base. The integration between Linux and SELinux relies on a security abstraction layer named Linux Security Modules (LSM), which enables the implementation of MAC policies as loadable kernel modules without the need to perform custom and invasive kernel modifications. LSM allows modules to mediate access to kernel objects by placing hooks in the kernel code just ahead of access to them [26]. These hooks are scattered through-out the kernel and have been classified as task, program loading, file-system, IPC, module and network hooks [25]. A security module such as SELinux implements a part or all of them.

The SELinux main capability is to separate security access control decisions from their enforcement, where the Security Server takes the security access control decisions and the LSM hooks enforce them [17]. Furthermore, SELinux has a third component known as Access Vector Cache (AVC), which is designed to speed-up the access validation decisions. The AVC maintains a cache of decisions made by the Security Server for subsequent accesses [17]. Figure 4.4 shows a standard DAC virtualized environment and compares it with the TRESCCA Secure Hypervisor REE, which relies on SELinux to define specific access rules for VMs' resources (e.g. the trusted shared memory) and to enforce access only to the trustworthy parties.

Moreover, the interaction between the Cloud and the SELinux VMs for the TRESSCA Secure Hypervisor is based on svirt, an extension of the libvirt [12] library which adds support for MAC security policies for virtual machines. The VOSYS effort in this direction led to a publicly available guide on the Company's website, which describes how to use svirt to create SELinux virtual machines with KVM on a Cortex-A15 [18].

Lastly, during the development of the TRESCCA Secure Hypervisor, the Linux DAC policy and SELinux have been evaluated and benchmarked. The result of this effort is a scientific publication, which compares the performance impact that these two security solutions have in virtualized environment [20]. This comparison shows unexpected results, as the performance of more secure guest (SELinux based) is better than the VM run in a standard DAC environment. In the paper, the cause of this performance improvement has been discussed and identified in Access Vector Cache (AVC), the SELinux cache decision system.

Trusted boot

The first step of the system's chain of trust is performed during the boot procedure. In fact when the machine boots, the security configuration of the system is not yet in place, and as a result the system is vulnerable to attacks which target to replace the boot procedure or run untrusted OSes.

In order to minimize this risk, the TRESCCA Secure Hypervisor has been designed as a multi stage bootloader. The first stage bootloader, in particular, is a tiny program stored in a on-chip ROM along with the public key needed for the attestation of the second stage bootloader. Since the first stage bootloader is stored in a read-only memory, it can not be updated and it's therefore critical for the security of the system. Soon after the initialization of the key system components, it checks the integrity and boots the second stage bootloader which is located in an external non-volatile memory (e.g., flash memory). The second stage bootloader then loads the microkernel binary in the system's Secure World memory and boots it.

Soon after, the third stage bootloader, implemented as a Trusted Application inside the TEE is executed. Thereby, when the Secure World OS is up and running, a specific TA checks the integrity of the Non Secure World OS binary (i.e. the Linux kernel with KVM) and its bootloader (fourth stage). If this last security check is successful, the fourth stage bootloader runs the Non Secure OS and the system can be considered ready to run guest OSes. On the other hand, if only one of these checks fails, the boot process will be stopped and the TRESCCA platform will enter in a secure and not operational state. Figure 4.5 shows the TRESCCA Secure Hypervisor chain of Trust.

The Trusted boot process is the key technology for the attestation of the user space applications because it ensures the integrity of the chain of trust. The TRESCCA Secure Hypervisor is able to run in the TEE an

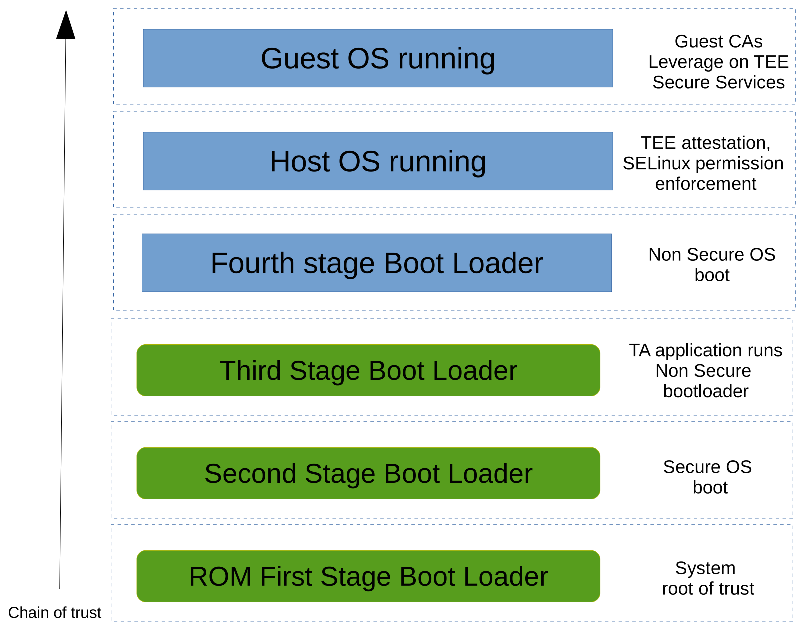


Figure 4.5: TRESCCA Secure Hypervisor Trusted boot procedure.

attestation service, which can check at any moment the integrity of its key components i.e., QEMU, libvirt, the VMs and their resources, etc. libvirt in particular, is extended to attest the VMs identity and integrity at each boot and in an event-driven manner, assuring to users and cloud administrators/providers the authenticity of the workloads run on the hardware. The security assets (fingerprints, keys, etc.) of these binaries are stored in the Secure World and by consequence can be updated frequently.

A trusted boot implementation such as what has been described above, is also part of the GlobalPlatform standard, as mentioned in Section 3.2.

4.3.2 Trusted communication

The communication between the two worlds and between the VMs and the host needs to be fast and secure. VMs shared memory mechanisms which provide high performance and low latency already exist for the KVM hypervisor [15, 19], but actually lack of TrustZone support or any other security related feature.

The Trusted Applications are able to read/write the content of the VMs shared memory, unless they know the address where the shared memory area begins because by design the Secure World is able to access the full Non Secure World address space. For this reason, the communication mechanims between TEE and REE needs to be extended to send the CA data memory address to the TA (Figure 4.3). In the REE, this mechanism needs to be secured in order to prevent attacks and information leakage. As described in above in Section 4.3.1, the TRESCCA Secure Hypervisor communication mechanism relies on SELinux to protect sensitive data.

On this regard, during the TRESCCA project, VOSYS developed vosyshmem [19], a shared memory mechanism for ARM virtual machines. Lastly, an interesting enhancement to the current implementation is the encryption of the shared memory area exploiting hardware accelerators as described in Section 2.2. This approach is not feasible in software, because the performance penalty of the data encryption and decryption is too high.

4.4 TEE Services

The TRESCCA Secure Hypervisor can be extended to support secure services such as VMs and SW attestation, introspection, cryptography, Random Number Generators (RNG), etc.. Each one of these extensions could be developed by the implementation of these components:

- A TEE application (or TA), leveraging on the capabilities of the TEE Internal API, which offers security sensitive functions only (secure storage, security algorithms, etc.).
- A guest client application (or CA) which implements the user interface and all the non sensitive functions.
- An optional host tool, which is needed for complex use cases where there is a need to interact with the hypervisor (e.g. for the VMs attestation).

These services are able to enforce the security reccomandations for the hypervisor and the VMs listed by the National Institute of Standards and Technology (NIST) Guide to Security for Full Virtualization Technologies [9]. Moreover, it is important to notice that, even if the implementations of TRESCCA TEE shown in Section 4.2.1 are quite different, the GlobalPlatform TEE APIs guarantee a certain degree of portability for the TEE services.

5 HYPERVISOR INTEGRATION WITH HSM-MEM AND HSM-NOC

HSM-Mem and HSM-NoC are hardware security modules that are compatible with a possible implementation of TEE, such as the one shown in Figure 5.1. Being both the HSM-mem and the HSM-NoC security sensitive hardware devices, their configuration and functions management has to be performed and controlled in an isolated environment. Thereby, the TRESCCA Platform's TEE is the best suited compartment to handle these tasks. As a consequence, each time that the TRESCCA Secure Hypervisor wants to change a rule in the HSM-NoC or allocate a new protected memory area through the HSM-mem, it interacts with the TEE compartment, asking to the secure operating system to complete the execution of the requested operation. Section 4.2 described how the KVM hypervisor and the virtual machines are able to interact with the Secure World and request security sensitive operations.

The technologies developed during the TRESCCA project are able to protect each virtual machine entirely, both with the encryption provided by the HSM-mem and with the access control rules provided by the HSM-NoC. As a matter of fact, an extension of the current HSM-NoC firewall can be used to set specific rules which enable virtual machines to securely share memory areas, but at the same time enhance the isolation of the VMs resources. Similarly the entire guest memory area could benefit from the HSM-mem to secure the VM memory content. However, such a high proctection would lead to important performance issues. For this reason, the approach taken by the TRESCCA project is to protect selected parts of the VMs, in order to benefit from the security provided by the HSM-NoC and HSM-mem while achieving a good performance level.

In the following sections, the interaction of the HSM-mem and HSM-NoC with the KVM hypervisor will be detailed.

5.1 HSM-NoC

5.1.1 Linux Driver Development and Validation

The objective of the coarse-grain NoC Firewall module (HSM-NoC) is to control access to system memory regions or to any memory-mapped slave peripheral. When integrating the NoC Firewall at the NoC initiator network interface the non-conforming transactions are not allowed to enter the NoC, thus saving throughput and energy. Alternatively, when placing a NoC Firewall after the target network interface, this offers a single point of protection for the target, thus avoiding consistency issues and synchronization overheads of updating the protection rules across different NoC Firewall units. In the latter case, if the access rules for a logical partition change, then all NoC Firewall configurations must be updated, while safeguarding against transactions that may be on the fly. The number of memory partitions, called segments, the size of each segment, and the properties or rules that are enforced for each segment are system-defined and programmable during operation (see Section 2.3). The NoC Firewall can be customized at design time to fit the requirements of a particular use case, in terms of number of segments and their size.

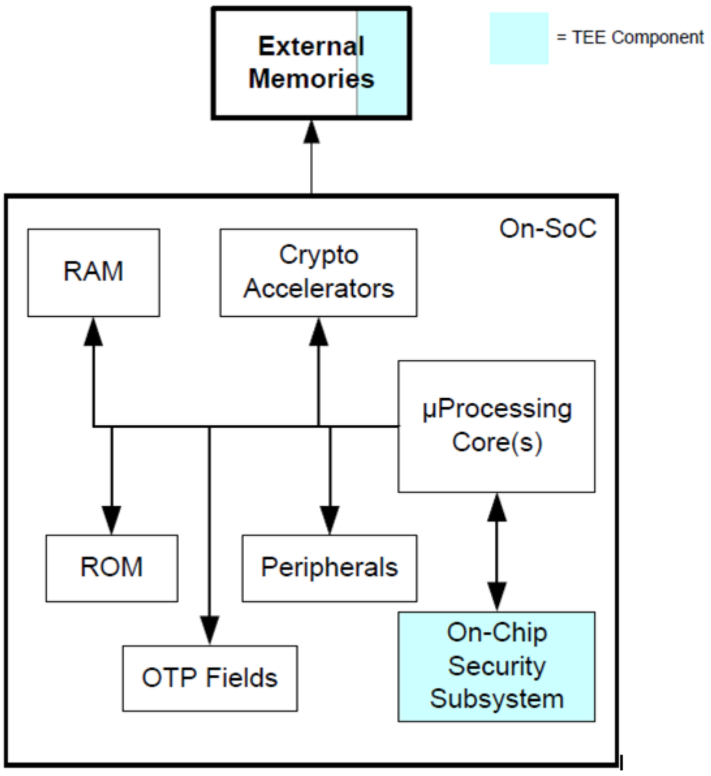


Figure 5.1: Realization of TEE; HSM-Mem and HSM-NoC are instances of the On-Chip Security Subsystem.

The NoC Firewall Architecture

The NoC Firewall uses the physical address of all incoming transaction requests to search for any preconfigured rules and determine the access control policy to apply. More specifically,

- If there is no match in the NoC Firewall data structures, then the transaction request is allowed to proceed.
- If an incoming address lies in-between a preconfigured address range, then the address belongs to a particular segment and therefore the rules table is accessed to decide if this transaction is compliant with the preset rules for this segment.

The coarse-grain protection implementation currently supports sixteen (16) memory segments. The maximum number of segments that can be implemented (without affecting timing) is technology-dependent. The 20-bit high order part of the physical address is utilized to concurrently search the range of each programmed segment. If successful, a corresponding enable signal for this segment is asserted and used to index to the discovered rule inside the set of rules; in a new implementation of the new NoC Firewall version, this indexing process can also use an optional 6-bit process identifier (PID), so that the indexing scheme points to the base address of the rules for a particular PID. Each rule in the implementation consists of eight non-encoded bits, hence rule memory is of size 1024 x 8. Each 8-bit rule in the implementation is formed by specifying the subfields: read, write, data, execute, privileged, non-privileged, secure, and non-secure domain. However, notice that not all subfields can be set independently.

Running Linux on the Zedboard fabric with the NoC Firewall ported on the FPGA

On the Zedboard FPGA, the NoC Firewall was implemented on top of an AMBA AXI4-Lite bus with "Software reset" and "User logic software register" options. This provides register base interaction between a master (ARM Core) and the TEI NoC Firewall which is viewed as a memory mapped device by the system. More specifically the device status configuration and control registers were mapped to a byte-addressable physical address space as follows:

- 0x40000000 – 0x40000004; this address maps to a 32-bit register used to reset the interrupt (when it is enabled)
- 0x40001000 - 0x40001080; this address region maps to 16 pairs of 32-bit registers. Each pair (START, STOP) defines the upper (START) and lower range (STOP) of the protected address segment. This segment must be defined as a multiple of a memory page, i.e. 4K.
- 0x40002000 – 0x40002040; this address region maps to 16 32-bit registers, whereas each register's 8 least significant bits will contain the rule for the range defined by the corresponding pair of registers (START, STOP) defined above.

In addition, to perform read/write accesses to the 512 MB of physical memory (DDR3) through the NoC Firewall device (in the programmable logic) a special mapping is used from physical address 0x80000000 to 0xA0000000, over an AXI4 interface. The device also uses a non shared interrupt line with number 91 to connect the device to the processing system's interrupt controller. Non shared interrupts were chosen for reasons of simplicity and since there was no issue with the total number of the IRQ lines. A number of tools were used for the installation of Linux and developing the NoC Firewall drivers [28, 29, 30]. More specifically, the Xilinx ISE design suite 14.7, the Xilinx Platform Studio (XPS) with the build-in Xilinx Synthesis Tool (XST) was used to synthesize, implement the NoC Firewall design in VHDL, and generate the bitstream file in order to be able to program the FPGA. Then, the Xilinx SDK was used for the creation of the First Stage Bootloader (fsbl), while the cross-compiler of the SDK was used for building the uboot (universal boot loader) from Xilinx u-boot-xlnx-xilinx-v14.7 (with option zynq_zed_config) and the kernel itself. More specifically, in order boot Linux on Zedboard, TEI has formatted an SD card with two partitions:

- Linux 3.17 kernel patched with Xilinx sources (uImage), BOOT.bin, and device tree blob (DTB) file on the first partition. The BOOT.bin file was generated from Xilinx SDK bootgen tool from the bitstream.bit, fsbl.elf, and uboot.elf (see above), while the DTB file is compiled from the default device tree source (DTS provided with the Linux kernel for the Zedboard, this data structure removes significant part of the hardware description out of the kernel binary).
- Linaro Ubuntu Linux file system from Xilinx on the second partition.

5.1.2 Implementation of the Linux Driver

TEI's aim in developing the Linux drivers for the NoC Firewall device was to evaluate the performance level of the TEI firewall implementation in a real system running GNU/Linux. As explained in the previous Section, most of the driver implementations are dependent on the embedded system design for which they were developed and the EDA tools used. However, the concepts on Linux driver development introduced in this Section are quite general and focus on

- configuring and communicating with the NoC Firewall device according to the specifications,

- supporting specific commands to perform read/write requests in order to access data or instructions via the NoC Firewall while considering different operating modes, and
- handling interrupts generated by deny rules.

The Linux drivers of the NoC Firewall are in the form of loadable kernel modules which can be attached to the kernel at runtime. Since the NoC Firewall is viewed under Linux as a memory mapped device, the kernel-space Linux drivers of the NoC Firewall are written as character devices and all I/O operations are memory mapped. Thus, CPU cores can access memory using virtual address pointers much more efficiently and without special-purpose instructions, while the compiler has much more freedom in register allocation and address-mode selection when accessing memory. The basic functions used for the design of the TEI character driver are as follows. During initialization of the NoC Firewall kernel module, the program performs the following functions:

- allocate the memory regions of the global physical address space used by the TEI device using function `struct resource *request_mem_region(unsigned long start, unsigned long len, char *name)`. Notice that the function allocates a memory region of len bytes, starting at start. If successful, the specified I/O memory allocation will be listed in `/proc/iomem`.
- map physical addresses to virtual ones, so that they can be accessed within the kernel, using two functions: a) `void *ioremap(unsigned long phys_addr, unsigned long size)` and b) void `*ioremap_nocache(unsigned long phys_addr, unsigned long size)`. These functions obtain a virtual pointer to the start of the physical memory, whereas `ioremap_nocache` does not use the cache, This is especially important when the cache system supports write-back.
- register the interrupt using the function `int request_irq (unsigned int irq, irq_handler_t handler, unsigned long irqflags, const char *devname, void *dev_id)`. This function allocates interrupt resources and enables the interrupt line and IRQ handling.

Notice that in order to enable support for sharable IRQ lines in the TEI NoC Firewall Driver, the kernel module sets in the `irqflags` bits the `SA_SHIRQ` bit. This bit signals that the interrupt can be shared between devices. Also the `dev_id` argument must be a unique device pointer address, for each device driver.

For reason of simplicity and since there was no problem with the number of the irqs selected to be used as non sharable. During normal operation, the following functions will be invoked:

- the standard pair of functions `unsigned int ioread32(void *addr)` and `void iowrite32(u32 value, void *addr)` to access any specific memory mapped register (e.g. NoC Firewall (START, STOP) segment or rule definitions). The address `addr` is the address obtained from `ioremap_*` (perhaps with an integer offset). The return value of the `ioread32` is what was read from I/O memory, while value is the value written to memory during `iowrite32`.
- the interrupt handler `irq_handler_t handler(int irq, void *dev_id, struct pt_regs *regs)` to clear any interrupt raised by the NoC Firewall device; notice that irq is the number of the irq line for the device, i.e. 91. Notice that there is a need to take care of initializing the hardware and setting up the interrupt handler beforehand.

Finally, upon exiting the module, the following functions are called:

- `void release_mem_region(unsigned long start, unsigned long len)` and `void iounmap(void * addr)` to free memory, and
- `void free_irq(unsigned int irq, void *dev_id)` to free resources related to the interrupt allocated with `request_irq()`.

5.1.3 KVM Interfacing with HSM-NOC

Full virtualization adds extra layers which require additional security management controls. Although virtualization helps in reducing the impact of a vulnerable service, dynamically combining many operating systems onto a single physical computer makes virtualized services or applications at least as insecure. More specifically, KVM can utilize the NoC Firewall drivers when accessing files/directories or copy/paste buffer and other resources on the host OS or another guest OS, to support

- **strong isolation (sandboxing)** of a Guest OS from other guests or the host OS (or the hypervisor from the host OS) when sharing hardware resources, i.e. CPU instructions, memory or storage. This is performed by partitioning these resources so that each guest OS can access shared resources according to static (pre-configured at compile time) or dynamic (runtime) rules. This resource provisioning prevents unauthorized access to resources (e.g. reading or writing data or infecting files) or injection, execution and propagation of malware code (virus) by a malicious Guest OS into another guest OS's memory.
- **partitioning of the global physical address space** in a memory-mapped system, thus enabling stronger security through performance isolation, in particular reducing the risk of denial of service attacks (e.g. excess resource consumption) by a guest OS.
- **mitigation of side-channel attacks**. These attacks exploit the physical properties of hardware to reveal information about resource utilization patterns. NoC Firewall can help in this direction by protecting accesses to sensitive information.

As described in the introduction, the best suited place for the HSM-NoC driver is the TEE. In this case, the HSM-NoC driver can be used by KVM hypervisor through a TA installed in the Secure compartment, in order to manage VMs system resource isolation (CPU, memory, and I/O accesses). Nonetheless, the HSM-NoC, is also able to protect the hypervisor itself, by configuring at the Non-Secure operating system boot time the HSM-NoC to protect the Hypervisor memory area. This would shrink considerably its attack surface, thereby enhancing the security of the virtualized system.

5.2 HSM-mem

Deliverable D3.1 *Security module software driver* describes the software stack for the HSM-mem:

- OS-independent, low-level, software driver,
- Software driver for the MutekH OS,
- Software Security Manager library (`libssm`),
- User Application Programming Interface (API)

The reader interested in the low-level details of the software stack for the HSM-mem will find them in D3.1. The development of a Linux version of the software components will be part of the WP4 activities but the principles and interfaces will be very similar to that of the MutekH version. The Linux version will be used for demonstration purpose. It could also allow the REE to interact directly with the HSM-mem and to manage the security of its memory pages without the intervening of the TEE. However, this section describes an example integration with the hypervisor that follows another approach, where the HSM-mem is entirely managed from the secure world. Indeed, the HSM-mem and its companion software are sensitive assets. Compromising them would allow an attacker to completely void the protection of the external memory and to perform sensitive data recovery or memory injection without being detected. Controlling the HSM-mem from the secure world reduces the attack surface and thus the probability that a software exploits compromises the external memory protection. In order to implement this scenario the HSM-mem software stack should be ported to the software stack used for the secure world (Fiasco.OC or OPTEE OS in this document). Again, the principles and interfaces would be very similar to that of the MutekH version.

The HSM-mem and its driving software are involved in various situations. In the following we present these situations, the role of the HSM-mem and of its driving software in an ideal TRESCCA platform. We also give preliminary indications about the possible implementation on a ZedBoard[31] prototyping board.

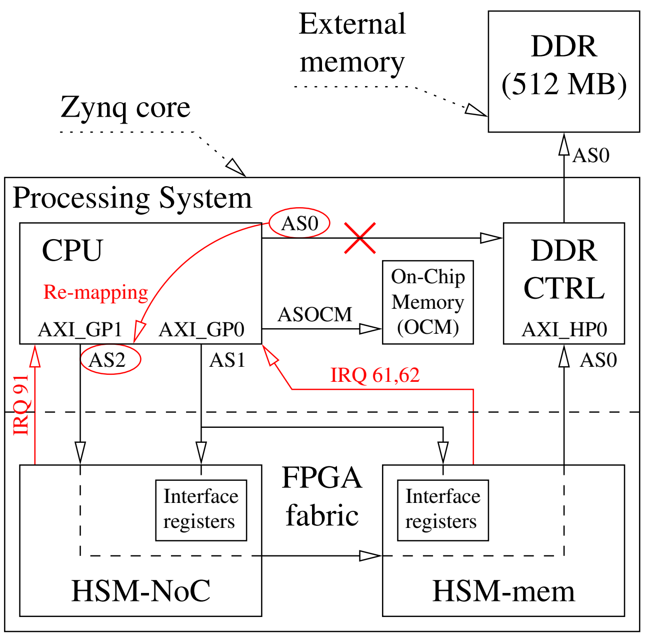
5.2.1 The ZedBoard prototype, global view

In the current version of the hardware prototype on the ZedBoard, the HSM-mem is mapped in the FPGA fabric of the Zynq core and the memory mapping of the various software components (boot loaders, Linux kernel...) is modified: the `[0...1G[` memory range (address space of the external memory from the CPU point of view) is shifted to `[2G...3G[` (address space of the FPGA fabric from the AXI_GP1 master port). Thanks to this shift, all external memory accesses from the CPU flow through the FPGA fabric, that is, the HSM-mem. The HSM-mem forwards the memory accesses to the DDR controller through the High Performance AXI port AXI_HP0. The `[1G...2G[` range (address space of the FPGA fabric from the AXI_GP0 master port) is used to access the interface registers of the HSM-mem for configuration, atomic operations, status reading... Figure 5.2.1 illustrates this set up. Note that, because the ZedBoard embeds only 512 MB of external memory, only the low half of the `[0...1G[` and `[2G...3G[` ranges are actually used. The interface registers of the HSM-mem are mapped in the `[0x40010000 - 0x40020000[` range. This memory mapping is fully compatible with the HSM-NoC and allows a straightforward integration, as illustrated on figure 5.2.1: the HSM-mem is integrated between the HSM-NoC and the external memory controller and shares with it the `[1G...2G[` range for configuration. The interfaces of the two HSMs are AXI4 for the regular memory accesses and AXI4-lite for the configuration. They can thus be plugged without any modification. The HSM-mem has also two output interrupt lines, one to signal integrity violations or access to invalid PSPE/SP (`irq_e`) and one to signal the completion of the HSM-mem atomic commands (`irq_c`). These two output interrupt lines are connected to the interrupt request lines 61 and 62 of the ARM core (while the HSM-NoC uses interrupt request line 91).

5.2.2 The secure boot sequence on the ideal TRESCCA platform

At boot time the HSM-mem is responsible for preventing malevolent sniffing and tampering of the external memories. This is a part of the secure boot sequence. The boot ROM code (true first stage boot loader) is thus responsible for initializing the HSM-mem and for configuring the first Security Policies (SPs) and Page Security Parameter Entries (PSPEs) to protect the external memory pages in which it will install the Second Stage Boot Loader (SSBL). It does so thanks to the low-level driver which code is entirely contained in the on-chip ROM. The boot ROM code (including the routines of the low-level HSM-mem driver it calls) runs only using the internal RAM because the external memory cannot be trusted until the HSM-mem initialization and configuration complete.

The boot ROM code loads the SSBL from an external mass storage (e.g. a Hard Disk Drive, a SD card or a QSPI on-board flash), installs it in the protected external memory pages, checks its integrity against a trusted reference digest and jumps into it. The trusted reference digest cannot be hard-wired in the boot ROM code or stored in on-chip ROM because this would make software updates of the SSBL impossible. Authentication



AS0: `[0...1G[`
AS1: `[1G...2G[`
AS2: `[2G...3G[`
ASOCM: `[4G-256K...4G[`

Figure 5.2: Example HSM-mem prototype on a ZedBoard.

using public key cryptography would not be sufficient to prevent SSBL downgrade attacks[1]. As a consequence, the trusted reference digest must either be stored in on-chip Non-Volatile Memory (NVRAM) or loaded from a remote trusted Certification Authority (CA). In the first case software updates are possible and just require an update of the digest in NVRAM. As the digest is stored on-chip, according the TRESCCA threat model, it cannot be tampered with. Downgrade attacks are detected because the digest of the outdated SSBL does not match the more recent digest in NVRAM. The only drawback is that the manufacturing process of the System-on-Chip (SoC) does not always support NVRAM. In the second case (CA) the boot ROM code must be complex enough to initiate a secure communication channel with the CA, that is, embed the driver of a smart card reader or a minimal secured network stack. The same properties hold: the SSBL can be updated and the new digest stored by the CA, the CA being remote and trusted, the digests cannot be tampered with, and downgrade attacks are detected because the digest of the outdated SSBL does not match the more recent digest of the CA. The drawback is the increased complexity of the boot ROM code, that is, the higher probability of security vulnerabilities and a larger memory foot print.

Once the SSBL is running, it continues the boot sequence according the same principles: allocate and protect external memory pages, load and check the next stage, hand off, and so on until the full software stack is up and running. Of course, in order to mitigate the performance impact, only memory pages that need protection are protected. And they are protected using the lightest policy satisfying the security requirements.

[1] A downgrade attack consists in replacing the last version of the SSBL by an older, superseded, one. As the outdated version also comes with its old valid signature, the attack cannot be detected by a simple signature check.

5.2.3 The secure boot sequence on the ZedBoard prototype

On the ZedBoard prototype the 256 KB On-Chip Memory (OCM) plays the role of the on-chip ROM and on-chip RAM. Like on the ideal TRESCCA platform the boot ROM code is stored on-chip, in a ROM that is temporarily mapped in the first 64 KB of the memory space. It initializes the Zynq core, detects the boot medium from the configuration of the on-board jumpers, loads the next stage (which, in the Xilinx-Zynq terminology, is named First Stage Boot Loader or FSBL) from the boot medium (e.g. a SD card or a QSPI on-board flash), installs it in the OCM, remaps the first 64 KB of the memory space to true memory and jumps into the FSBL. After remapping of the first 64 KB of the memory, the content of the boot ROM is not accessible any more. This is how Xilinx protects the industrial property of the boot ROM code.

This start up is almost the one of the ideal TRESCCA platform. The only deviation is that the boot ROM code being fixed and secret, it cannot be modified to initialize the HSM-mem and protect external memory pages for the next stage. So, in the ZedBoard case, it is the FSBL that will embed the HSM-mem low-level driver and play the role of the boot ROM code of the ideal TRESCCA platform. It will be loaded in OCM by the Zynq boot ROM code and will stay there as if it was in ROM. A portion of the OCM will thus not be available for the other software components but this is perfectly acceptable, the OCM being much larger than what is commonly found in SoCs. Moreover, thanks to the e-Fuses and the cryptographic accelerators of the Zynq core, it is possible to check the integrity of the FSBL when loading it and to abort the boot sequence if the check fails. The combination of the boot ROM code and the FSBL of the ZedBoard could thus perfectly play the role of the real boot ROM code of the ideal TRESCCA platform.

Another slight deviation of the ZedBoard-based demonstrator will be the source of the trusted reference digest against which the SSBL integrity will be checked. While it would be perfectly possible, the FSBL will not communicate with a CA. Developing the FSBL would be more complex and error prone and it would not bring any innovation to the project. As the Zynq core does not embed NVRAM[2] neither, the FSBL will have the SSBL digest hard-wired. This apparent limitation of the prototype is not a real one because it could very easily be removed from a real platform.

On the ZedBoard the FSBL will run using only the OCM, both for its code and data, as would be done on the ideal TRESCCA platform. It will initialize the HSM-mem using the routines of the low-level driver. Next, it will prepare the memory protection of a set of external memory pages, load the SSBL into these pages, check its integrity against the hard-wired reference digest and jump into it.

In the ZedBoard prototype the SSBL will be a customized version of U-Boot[5]. U-Boot is a very good representative of the kind of SSBL that could be found on the ideal TRESCCA platform. U-Boot is powerful enough to initiate a secure network communication with a CA from which it can load a reference digest for the next stage. When loading the next software stage (either from the network or from mass storage) U-Boot can again check its integrity against the certificate. During all its execution it will use only the OCM or the external memory pages protected by the HSM-mem.

In summary, the FSBL and the SSBL (U-Boot) will be customized to use the low-level HSM-mem driver to initialize the HSM-mem and to create the SPs and PSPEs necessary to protect the sensitive external memory pages. Depending on the specific target application the protection of the last stage of the software stack can be different. In the case of the KVM TEE it can, for instance, protect the whole secure world (tiny trusted OS and Trusted Applications), plus, on demand of the Rich OS, some memory pages shared between the REE and the TEE.

[2]Except the e-Fuses (that cannot really be considered a replacement because they are in limited number) and a battery-backed static RAM (that could somehow emulate an internal NVRAM but would require a modification of the ZedBoard to add the battery).

5.2.4 The run time in the secure world (TEE)

At run time the HSM-mem is used to protect the sensitive external memory pages either in confidentiality, integrity or both. The Software Security Manager (SSM) manages the HSM-mem to create and delete SPs and PSPEs dynamically according the needs. It is built on top of the Software Security Manager library (`libssm`). The `libssm` itself relies on the OS driver of the HSM-mem. Finally, for performance and security reasons, the OS driver uses the routines offered by the OS-independent, low-level, driver stored in the on-chip ROM (a part of the OCM in the ZedBoard prototype). The SSM is tightly coupled to the memory manager of the OS.

In the following part of this section, the HSM-mem integration with the TEE is explained using the FIASCO.OC microkernel. However the same approach could be used with OPTEE OS as well. In the Fiasco.OC/L4Re TEE case, the SSM shall be integrated to the Sigma0 root pager and Moe root task of the Fiasco.OC microkernel because these two services are responsible for the memory management in the secure world (TEE). This integration shall be done by adapting the memory management, probably as an extension of the `l4re_ma_flags` enumeration type used to specify properties of the allocated pages (see listing 5.1).

```
enum l4re_ma_flags {
  L4RE_MA_CONTINUOUS        = 0x01,
  L4RE_MA_PINNED            = 0x02,
  L4RE_MA_SUPER_PAGES       = 0x04,
  L4RE_MA_READ_ONLY         = 0x08, // Read-only page
  L4RE_MA_INTEGRITY         = 0x10, // Integrity protection
  L4RE_MA_CONFIDENTIALITY = 0x20, // Confidentiality protection
};
```

Listing 5.1: Modified `l4re_ma_flags` Fiasco.OC enumeration type

Of course, the allocator shall also be modified to make use of these new flags and call the appropriate routines of the `libssm`. It shall also handle data structures to keep track of the created SPs and decide when to reuse them for a newly allocated page or when to delete them. When deallocating pages, the deallocator must also disable the corresponding PSPE and, if needed, delete the associated SP.

The ELF loaders shall also be modified in order to request the appropriate security policies for the initial environment and the data spaces they create for the loaded applications. The modifications consist in interpreting a HSM-mem dedicated ELF section that associates security policies to linkage sections when it is part of the ELF, else, the loader must apply a default policy. The definition of the default policy is platform dependent.

Finally, the HSM-mem user API shall be used by all HSM-mem aware applications running in the secure world when they request the allocation of memory pages, to specify a security policy for the allocated pages. HSM-mem unaware applications shall be applied a default policy, which is again platform dependent.

With these minor modifications the TEE shall be able to protect its own memory pages according its security requirements.

5.2.5 Interaction with KVM and the non secure world (REE)

The TEE is virtualized such that the non-secure world can access the services it offers. This is done thanks to the TEE client API and the REE communication agent. In order to expose the HSM-mem to the non-secure world, the same adaptations needed for the TEE must be done to the REE software stack: the user API must allow a CA to specify the security policy it wants to be applied to a requested memory region. This security requirement (read-only or read-write, integrity protection or not, confidentiality protection or not) must traverse all software layers until they reach the TEE where they can be handled properly by the SSM. Thanks to these enhancements the memory regions used by the REE can benefit the HSM-mem protection. Of course, this can be used by the REE hypervisor (Linux+KVM) for its own memory pages, by the hosted VMs or the CAs. It can (and should) also be used to protect the memory regions shared between the REE and the TEE.

6 CONCLUSION

This document describes the KVM based TRESCCA Secure Hypervisor, its hardware and software components as well as the extensions needed by the hypervisor and the VMs to empower the TRESCCA platform.

A thoughtful state of the art study performed in the design phase has been presented in the first part of the document. It has shown the limits of the existing solutions, but most importantly, has demonstrated the need of a virtualized implementation of security a API such as GlobalPlatform TEE.

Moreover, different implementations of the Trusted Execution Environment have been explored, hardware and software wise, with the intent of finding the best solution for each scenario in which the TRESCCA platform could operate. In fact, both TrustZone and HSM-NoC could be used to implement in hardware the trusted compartment, which can offer services to the VMs either through a microkernel or a thin library. These alternative implementations have been detailed, highlighting benefits and caveats of each solution.

Lastly, the integration the HSM-NoC and HSM-mem in the TRESCCA virtualized environment is detailed in the last chapter, emphasizing the VMs benefits given by the HW modules developed during the project.

The result is thus the KVM based TRESCCA Secure hypervisor, a SW/HW solution for securing the VMs and their applications, which offers a virtualized implementation of the TEE and SELinux isolation, as well as the HW security capabilities of TrustZone, HSM-NoC and HSM-mem. In addition, the TRESCCA Secure hypervisor leverages on the TEE services to enable novel security functions, which combined with the support for the hybrid migration developed by VOSYS and detailed in D3.4, pave the way to a completely new Cloud Computing infrastructure, more secure and focused on the user.

ACRONYMS

API	Application Programming Interface
BIOS	Basic Input / Output System
CA	Client Application
GB	Giga Byte (1024 MB)
HSM	Hardware Security Module
HSM-mem	Hardware Security Module for memory protection (confidentiality, integrity)
HSM-NoC	Hardware Security Module for Network on Chip protection (access control, *firewalls*)
kB	kilo-Byte (1024 Bytes)
KVM	Kernel-based Virtual Machine
MMU	Memory Management Unit
NoC	Network on Chip
OS	Operating System
PaaS	Platform as a Service
RAM	Random Access Memory
REE	Rich Execution Environment
SaaS	Software as a Service
STNoC	STMicroelectronics Network on Chip
SoC	System on Chip
SSL	Secure Socket Layer
TA	Trusted Application
TCG	Trusted Computing Group
TEE	Trusted Execution Environment
TPM	Trusted Platform Module
TXT	Trusted eXecution Technology
VM	Virtual Machine

BIBLIOGRAPHY

[1] Joakim Bech. "`https://www.linaro.org/blog/core-dump/op-tee-open-source-security-mass-market/`".

[2] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[3] Gary Chen. Kvm – open source virtualization for the enterprise and openstack clouds, 2014.

[4] Christoffer Dall and Jason Nieh. Kvm/arm: Experiences building the linux arm hypervisor, 2013.

[5] Denx. "`http://www.denx.de/wiki/U-Boot`".

[6] Ivan Gonzalez, Sergio Lopez-Buedo, and Francisco J Gomez-Arribas. Implementation of secure applications in self-reconfigurable systems. *Microprocessors and Microsystems*, 32(1):23–32, 2008.

[7] Trusted Computing Group. Tpm mobile with trusted execution environment for comprehensive mobile device security, 2012.

[8] heartbleed. `http://www.heartbleed.com`.

[9] P Hoffman, K Scarfone, and M Souppaya. Guide to security for full virtualization technologies. *National Institute of Standards and Technology (NIST)*, pages 800–125, 2011.

[10] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[11] ksys labs. "`https://github.com/Ksys-labs/L4Reap`".

[12] libvirt website. "`http://www.libvirt.org`".

[13] ARM Limited. Arm security technology, 2005-2009.

[14] Antonio Lioy, Gianluca Ramunno, and Davide Vernizzi. Trusted-computing technologies for the protection of critical information systems. In *Proceedings of the International Workshop on Computational Intelligence in Security for Information Systems CISIS'08*, pages 77–83. Springer, 2009.

[15] Cam Macdonell, Xiaodi Ke, Adam Wolfe Gordon, and Paul Lu. Low-latency, high-bandwidth use cases for nahanni/ivshmem, 2011.

[16] Svetlin A Manavski. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. In *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, pages 65–68. IEEE, 2007.

[17] Frank Mayer, David Caplan, and Karl MacMillan. *SELinux by Example: Using Security Enhanced Linux*. Pearson Education, 2006.

[18] Virtual Open Systems Michele Paolino. "`http://www.virtualopensystems.com/en/solutions/guides/kvm-svirt-omap5/`".

[19] Michele Paolino. A shared memory zero-copy mechanism for arm vms: vosyshmem. `http://www.virtualopensystems.com/en/products/vosyshmem-zerocopy/`, 2014.

[20] Michele Paolino, Mian M. Hamayun, and Daniel Raho. A performance analysis of arm virtual machines secured using selinux. In Frances Cleary and Massimo Felici, editors, *Cyber Security and Privacy*, volume 470 of *Communications in Computer and Information Science*, pages 28–36. Springer International Publishing, 2014.

[21] Z Cliffe Schreuders, Tanya McGill, and Christian Payne. Empowering End Users to Confine their own Applications: The Results of a Usability Study Comparing SELinux, AppArmor, and FBAC-LSM. *ACM Transactions on Information and System Security (TISSEC)*, 14(2):19, 2011.

[22] GlobalPlatform Device Technology. Tee client api specification, 2010.

[23] GlobalPlatform Device Technology. Tee internal api specification, 2011.

[24] GlobalPlatform Device Technology. Tee system architecture, 2011.

[25] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux Security Module Framework. In *Ottawa Linux Symposium*, volume 8032, 2002.

[26] Chris Wright, James Morris, Greg Kroah-Hartman, Crispin Cowan, and Stephen Smalley. Linux Security Modules: General Security Support for the Linux Kernel. In *Foundations of Intrusion Tolerant Systems (OASIS'03)*, page 213. IEEE Computer Society, 2003.

[27] XV6. `http://pdos.csail.mit.edu/6.828/2014/xv6.html`.

[28] zedboard.org. `http://zedboard.org/content/zedboard-create-planahead-project-embedded`

[29] zedboard.org. "`http://zedboard.org/sites/default/files/blogger_importer/08/zedboard-sdk-helloworld-example.html`".

[30] zedboard.org. "`http://zedboard.org/content/creating-custom-peripheral`".

[31] zedboard.org. "`http://zedboard.org`".