Master's Thesis

Supporting Software-based TPM Emulator in ARM Virtualization Environment

Mingyuan Gao

Department of Computer Science and Engineering

Pohang University of Science and Technology

2013


POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

(ARM 가상화 환경에서 소프트웨어기반 TPM 에뮬레이터 지원 방법)

Supporting Software-based TPM Emulator in ARM Virtualization Environment

Supporting Software-based TPM Emulator in ARM Virtualization Environment

by

Mingyuan Gao
Department of Computer Science and Engineering
Pohang University of Science and Technology

A thesis submitted to the faculty of the Pohang University of Science and Technology in partial fulfillment of the requirements for the degree of Master of Science in the Computer Science and Engineering

Pohang, Korea
06. 20. 2013
Approved by
Chanik Park (Signature)
Academic Advisor

Supporting Software-based TPM Emulator in ARM Virtualization Environment

Mingyuan Gao

The undersigned have examined this thesis and hereby certify that it is worthy of acceptance for a master's degree from POSTECH

06. 20. 2013

Committee Chair Chanik Park (Seal)

Member Jong Kim (Seal)

Member Sungjoo Yoo (Seal)

MCSE Mingyuan Gao
20111047 Supporting Software-based TPM Emulator in ARM Virtualization Environment, ARM 가상화 환경에서 소프트웨어기반 TPM 에뮬레이터 지원 방법 ,
Department of Computer Science and Engineering, 2013, 57p,
Advisor: Chanik Park,
Text in English.

ABSTRACT

As mobile technology matures, mobile devices (principally smart phones and tablets) are increasingly being used in both personal and corporate environments. While mobile devices bring great convenience to us, security problems also ensue. Current mobile devices lack the hardware-based root of trust features (e.g., Trusted Platform Module, or TPM) that are increasingly built into laptops, PCs and other types of hosts. Unfortunately, mobile devices are constrained in space, cost and power dimensions that make the use of a discrete TPM difficult.

Since a software-based TPM emulator can provide the same capabilities of a hardware TPM in terms of *TPM Commands*, thus, if 1) *the roots of trust of the TPM emulator can be securely handled*, 2) *an isolated secure environment can be provisioned in the mobile device, be it software environment or hardware environment*, without the presence of a hardware TPM, the approximate same capabilities of a hardware TPM can be obtained by running a software-based TPM emulator in the said secure environment. In this thesis, we assume the *roots of trust* of a software-based TPM emulator are securely handled. However, even with

the above assumption, how to provision the isolated secure environment is still a challenging task.

In this thesis, three approaches are proposed in the context of ARM virtualization environment, i.e., *Linux Container (LXC)-based Approach*, *Virtual Machine (VM)-based Approach*, and *Firmware TPM* (which is based on TrustZone Virtualization). *LXC-based approach* uses a LXC to protect the software-based TPM emulator, and *VM-based approach* uses a VM; these two approaches were implemented and evaluated. *Firmware TPM* refers to the alternate software implementation of the TPM specification in the context of a Trusted Execution Environment; the proposed reference architecture for *Firmware TPM* is given in the thesis, but its implementation is left as future work in that this approach requires far more programming effort than a master's thesis project.

The evaluation result showed that the software-based TPM emulator can provide the same capabilities of a hardware TPM provided that 1) the *roots of trust* of the emulator are securely handled, 2) the proposed secure environments (Linux Container or VM) are secure enough. Though the addition of a software-based TPM Emulator on a mobile device incurs some overhead on the system, the overhead is acceptable for modern mobile devices, which typically shipped with more than 1GB memory and 1GHZ or faster CPUs.

In the last, recommended optimizations for current work are provided, along with considerations on future implementation of *Firmware TPM*.

Contents

1 Introduction ……..…. 1

2 TPM and Software-based TPM Emulator ……..... 4

2.1 TPM ……..……... 5

2.2 TCG Software Stack …..... 8

2.3 Software-based TPM Emulator ……..….. 10

3 Proposed Secure Environments for Software-based TPM Emulator ……..... 12

3.1 Linux Container-based Approach ……..……..…. 14

3.2 Virtual Machine-based Approach ……..….. 18

3.3 Firmware TPM ……...….. 21

4 Implementations ……..... 24

4.1 Linux Container-based Approach ……..…..…….. 25

4.2 Virtual Machine-based Approach ……..……...…….. 27

5 Evaluation ……..…... 29

5.1 Capability Evaluation of Software-based TPM Emulator ……..…..……..… 29

5.2 Overhead Evaluation of Software-based TPM Emulator …...…….…… 31

5.3 Execution Speed Evaluation of Software-based TPM Emulator ……...…….. 33

6 Future Work ……..... 34

6.1 Optimizing Current Work ……..…….. 34

6.2 Future Implementation Considerations of Firmware TPM ……..... 35

7 Conclusion ……..... 37

8 Appendices 39

8.1 Introduction to Linux Control Groups 39

8.2 Introduction to Linux Namespaces 44

8.3 Capability Evaluation Result of Software-based TPM Emulator 47

Publications 57

References 58

Acknowledgement 60

List of Figures

Fig. 1 Architectural Overview of the TSS ……. 9

Fig. 2 Overview of Software-based TPM Emulator ……..….. 10

Fig. 3 Linux Container-based Approach ……..…….. 16

Fig. 4 Virtual Machine-based Approach ……..…….. 19

Fig. 5 Firmware TPM ……..…… 23

Fig. 6 Implementation of Linux Container-based Approach ……..……. 25

Fig. 7 Implementation of Virtual Machine-based Approach ……..….. 28

Fig. 8 cgroups Rule 1 ……..…… 41

Fig. 9 cgroups Rule 2 ……..…… 41

Fig. 10 cgroups Rule 3 ……..….. 42

Fig. 11 cgroups Rule 4 ……..…….. 43

List of Tables

Table 1 Evaluation Result of Commands Implemented in the TPM ……. 30

Table 2 Evaluation Result of Overhead of Software-based TPM Emulator ……..…. 31

Table 3 Evaluation Result of Execution Speed of Software-based TPM Emulator …...… 33

1. Introduction

A popular trend for hackers focuses on attacking the client [1]. In 1980s, hackers attacked the network by sniffing passwords and hijacking network sessions. As applications encrypted data going across the network, hackers began attacking servers by taking advantages of misconfigured or buggy services, like web servers. Companies responded to these attacks with firewall intrusion detection and security auditing tools to protect their servers. Thus, hackers have increasingly turned to unprotected clients, such as PCs, laptops and smart phones.

As mobile technology matures, mobile devices (principally smart phones and tablets) are increasingly being used in both personal and corporate environments. With mobile devices, we can access our bank accounts, and do Internet shopping and even Internet voting. Further, employees can access enterprise services, data and resources to carry out work-related activities [2].

While mobile devices bring great convenience to us, security problems also ensue. Current mobile devices lack the hardware-based root of trust features (e.g., Trusted Platform Module, or TPM) that are increasingly built into laptops and other types of hosts [2]. Mobile devices are also vulnerable to “jailbreaking” and “rooting”, which provide device owners with greater flexibility and control over the devices, but also bypass important security features which may introduce new vulnerabilities [2]. Even worse, with the increasing complexity of mobile Operating Systems (OSs), a growing number of PC-like malware is starting to surface, such as Trojans, Keyloggers [3]. Therefore, *it is particularly important to make mobile devices a secure platform*.

Nowadays, software alone cannot address the security problems, and hardware-backed technologies (such as TPM and Trusted Execution Environment, or TEE) are increasingly

being adopted. Unfortunately, mobile devices are constrained in space, cost and power dimensions that make the use of a discrete TPM difficult.

Since a software-based TPM emulator can provide the same capabilities of a hardware TPM in terms of *TPM Commands*, thus, if 1) *the roots of trust of the TPM emulator can be securely handled*, 2) *an isolated secure environment can be provisioned in the mobile device, be it software environment or hardware environment*, without the presence of a hardware TPM, the approximate same capabilities of a hardware TPM can be obtained by running a software-based TPM emulator in the said secure environment. In this thesis, we assume the *roots of trust* of a software-based TPM emulator are securely handled. However, even with the above assumption, how to provision the isolated secure environment is still a challenging task.

In this thesis, three approaches are proposed in the context of ARM virtualization environment, i.e., *Linux Container (LXC)-based Approach*, *Virtual Machine (VM)-based Approach*, and *Firmware TPM* (which is based on TrustZone Virtualization). *LXC-based approach* uses a LXC to protect the software-based TPM emulator, and *VM-based approach* uses a VM; these two approaches were implemented and evaluated. *Firmware TPM* refers to the alternate software implementation of the TPM specification in the context of a Trusted Execution Environment; the proposed reference architecture for *Firmware TPM* is given in the thesis, but its implementation is left as future work in that this approach requires far more programming effort than a master's thesis project.

The evaluation result showed that the software-based TPM emulator can provide the same capabilities of a hardware TPM provided that 1) the *roots of trust* of the emulator are securely handled, 2) the proposed secure environments (Linux

Container or VM) are secure enough. Though the addition of a software-based TPM Emulator on a mobile device incurs some overhead on the system, the overhead is acceptable for modern mobile devices, which typically shipped with more than 1GB memory and 1GHZ or faster CPUs.

The remaining of this thesis is organized as follows. Section 2 introduces TPM and software-based TPM emulator. Section 3 describes three proposed secure environments for software-based TPM emulator in the context of ARM virtualization environment. Section 4 elaborates the two implemented secure environments. Section 5 provides the evaluation results of the two implemented approaches. Section 6 provides recommended optimizations for current work, along with the considerations on future implementation of *Firmware TPM*. Section 7 concludes the thesis. Section 8 is the appendices that contain detailed explanation of the two exploited Linux kernel features: *cgroups* and *namespaces*, as well as the detailed evaluation result of capability of software-based TPM emulator.

2. TPM and Software-based TPM Emulator

Can software be made completely secure? Probably not. Modern OSs for smart devices are incredibly complex. For example, a recent Linux kernel alone has more than 15 million lines of source code; for the recent Linux-based Android releases, they are estimated to have billions of lines of source code. Several recent studies have shown that typical product-level software has roughly one security-related bug per thousand lines of source code across its lifetime [1].

Without hardware support, it is likely impossible to detect the presence of malicious code in a system [1]. So far, all attempts to detect malicious changes in software without hardware support have ultimately been circumvented. In contrast, with a little bit of hardware support, it is quite easy to detect compromise.

In terms of mobile device security, from the heart, we need to protect the most sensitive information, such as private and symmetric keys, from theft or use by malicious code. Trusted Computing Group's (TCG's) TPM was designed to achieve such a goal. The TPM has rather limited functionality, but that is an advantage when it comes to certifying that it works as designed [1]. And the design has been made deliberately flexible so that it can be applied to almost any problem that occurs in the security field.

2.1. TPM

The TPM [1] has been designed to protect security by ensuring the following:

- Private keys cannot be stolen or given away.
- The addition of malicious code is always detected.
- Malicious code is prevented from using the private keys.
- Encryption keys are not easily available to a physical thief.

The TCG chip accomplishes these goals with three main groups of functions, as follows:

- Public key authentication functions
- Integrity measurement functions
- Attestation functions

The *public key authentication functions* [1] provide on-chip key pair generation using a hardware random number generator, along with public key signature, verification, encryption, and decryption. By generating the private keys in the chip, and encrypting them anytime they are transferred outside the chip, the TPM guarantees that malicious software cannot access the keys at all. Even the owner of the keys cannot give the private keys away to phishing or pharming attacks, as the keys are never visible outside the chip unencrypted. Malicious code could use the private keys on the TPM, so some way needs to be provided to ensure that malicious code cannot use the keys either.

The *integrity measurement functions* [1] provide the capability to protect private keys from access by malicious code. In a trusted boot, the chip stores in Platform Configuration Registers (PCRs) hashes of configuration information throughout the boot sequence. Once booted, data (such as private keys) can be "sealed" under a PCR. The sealed data can be unsealed only if the PCR has the same value as at the time of sealing. Thus, if an attempt is made to boot an alternative system, or a virus has "backdoored" the OS, the PCR value

will not match and the unseal will fail, thus protecting the data from access by the malicious code.

The *attestation functions* [1] keep a list of all the software measurements committed to the PCRs, and can then sign them with a private key known only by the TPM. Thus, a trusted client can prove to a third party that its software has or has not been compromised.

Malicious programs, such as spyware and Trojans, will be detected by changes in the PCR measurement, which can then cause the TPM to refuse to unseal sensitive data, or to refuse to use private keys for signing or decryption. If vulnerable or misconfigured programs are exploited, any changes they make to files can be similarly detected. Any attempts to gain authentication secrets, such as by phishing or pharming, will fail, as the owner of the authentication private keys cannot give the keys away. Data encrypted under keys sealed by the TPM will be much harder to access in the case of theft, as the attacker would need to open up the chip to get its storage root key in order to be able to unseal the protected keys. While possible, this is really difficult and expensive. Similarly, encrypted communications are much more immune to eavesdropping, if the encryption keys are exchanged or stored by a TPM.

The TPM is intended to be a trustworthy device for certain low-level security services, including hardware-based credential storage, device identity, and software integrity [3]. In conjunction with trustworthy software, it is intended to serve as a trustworthy base for trusted infrastructure leading to software applications that are trusted.

In TPM, three roots of trust are provided [4]:

Root of trust for measurement (RTM) A trusted implementation of a hash algorithm, responsible for the first measurement on the platform - whether at boot time, or in order to put the platform into a special trusted state.

Root of trust for storage (RTS) A trusted implementation of a shielded location for one or more secret keys – probably just one, the Storage Root Key (SRK).

Root of trust for reporting (RTR) A trusted implementation of a shielded location to hold a secret key representing a unique platform identity, the Endorsement key (EK).

The SRK and EK use asymmetric cryptography, and the TPM protects the secret part of the key pair. The public part of the EK should be signed into a certificate by the manufacturer; and for SRK, it is established when someone takes ownership of the platform, and may be re-initialized (losing all the secrets it protects) when the platform passes to a new owner.

2.2. TCG Software Stack

The TPM is in essence a passive storage device that is hard mounted on the motherboard. The TPM is attached to the Low Pin Count (LPC) bus, which is also used for attaching the system BIOS flash memory. This ensures that the TPM device is available during the early system bootstrap before any other device is initialized.

Communication with the TPM is typically handled by the TCG Device Driver Library (TDDL), and its interface is defined by the TCG Software Stack (TSS) specification. This library typically communicates with a device driver inside the kernel, and this device driver communicates with the actual TPM device.

The entry point for any programmer writing a TPM-based application is the TSS. The TSS specification defines an architecture that makes accessing the TPM simple and direct, while exposing all the functionality that the TPM provides in a vendor-neutral way. The TSS also provides APIs for functionality on top of that provided by the TPM, such as the ability to store key objects persistently on disk, connecting with TPMs on both local and remote machines, and conversion of data blobs between portable formats (TSS 1.2 only).

The TSS [1] is composed of three logical components: the TDDL, the TCG core service (TCS), and the TCG service provider (TSP) (Fig. 1).

The TDDL [1] is a library that provides an API to interface with the TPM device driver. Generally, TPM vendors will ship a TDDL library along with their TPM device driver so that TSS implementers can easily interface with it. The TDDL library offers a small set of APIs to open and close the device driver, send and receive data blobs, query the device driver's properties, and cancel a submitted TPM command.

The TCS [1] layer has several jobs. It provides management of the TPM's resources, such as authorization session and key context swapping. It also provides a TPM command blob

generator, which converts TCS API requests into the necessary byte streams that a TPM understands. It provides a system-wide key storage facility and synchronizes application access from the TSP layer.

The TSP [1] layer is called directly by the application and is implemented as a shared object, or dynamic linked library. The TSP interface (TSPI) exposes all the TPM's capabilities and some of its own, such as key storage and pop-up dialog boxes for authorization data.

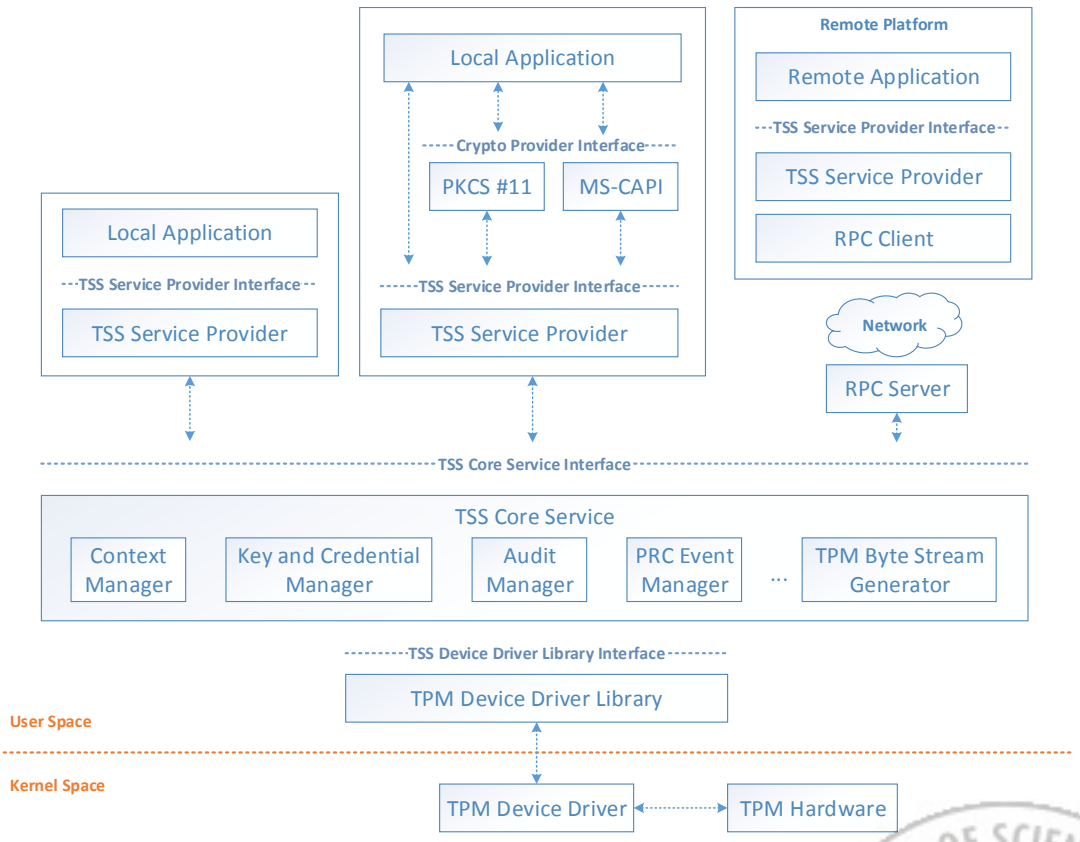


Fig. 1. Architectural Overview of the TSS

2.3. Software-based TPM Emulator

The objective of the TPM emulator project [5] is the implementation of a software-based TPM/MTM emulator as well as of an appropriate TDDL. The emulator provides researchers and engineers of trusted systems with a powerful testing and debugging tool that can also be used for educational purposes. Thanks to its portability and interoperability, the TPM emulator runs on a variety of platforms (including Linux, Mac OS X, and Windows) and is compatible with the most relevant software packages and interfaces.

The TPM emulator package comprises three main parts: a user-space daemon (*tpmd*) that implements the actual TPM emulator, a *tddl* as the regular interface to access the emulator, and a kernel module (*tpmd_dev*) that provides the character device */dev/tpm* for low-level compatibility with TPM device drivers (Fig. 2).

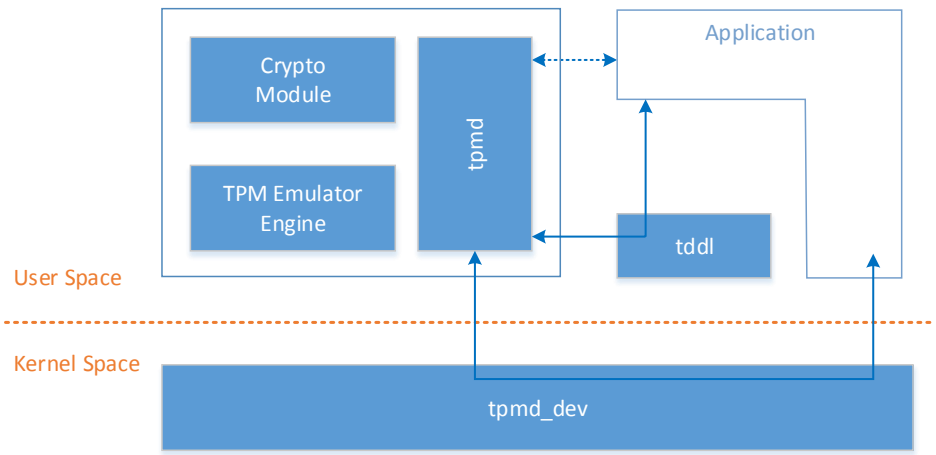


Fig. 2. Overview of Software-based TPM Emulator

TDDL and Kernel Module

The most convenient way to access a TPM is by means of the so-called TDDL as specified by the TCG. Applications that use the TDDL interface can be forced to use the TPM

emulator instead of a real TPM by simply exchanging this library. However, despite the existence of this well-defined and easy to use interface, several applications, tools and libraries rather directly access the TPM by its device driver interface, i.e., by the device file */dev/tpm*. Therefore, in order to be compatible with hardware TPMs at the lowest possible level, the TPM emulator package includes a Linux kernel module called *tpmd_dev*, which simulates a hardware TPM driver interface by providing the device */dev/tpm* and by forwarding all commands to the TPM emulator daemon.

TPM Emulator Daemon

The TPM emulator daemon implements the actual TPM emulator and consists of the daemon application, the TPM emulator engine, and the cryptographic module. After initializing the TPM emulator engine, the daemon application opens a UNIX domain socket and listens on it for incoming TPM commands. Upon the reception of a valid command, the request is processed by the TPM emulator engine and the corresponding response returned to the sender of the command.

3. Proposed Secure Environments for Software-based TPM Emulator

Since software alone cannot address the security problems, more and more hardware-backed technologies, such as TPM and TEE, has been proposed. Because the TPM chip can provide a system with *roots of trust*, which leads to trusted applications on the system, more and more TPM chips are increasingly built into laptops, PCs, Servers and other types of hosts.

Unfortunately, mobile devices are constrained in space, cost and power dimensions that make the use of a discrete TPM difficult. Since a software-based TPM emulator can provide the same capabilities of a hardware TPM in terms of *TPM Commands*, thus, if 1) *the roots of trust of the TPM emulator can be securely handled*, 2) *an isolated secure environment can be provisioned in the mobile device, be it software environment or hardware environment*, without the presence of a hardware TPM, the approximate same capabilities of a hardware TPM can be obtained by running a software-based TPM emulator in the said secure environment.

In the following text, we assume that the *roots of trust* of a software-based TPM emulator are securely handled, which can be aided by the device ROM and security subsystem of an ARM SoC. For example, Samsung Exynos 5250 Processor [6] provides hardware engines and memory for security, including 64 KB on-chip secure boot ROM for secure boot, 352 KB on-chip secure RAM for security function, hardware crypto accelerators (AES, DES/3DES, ARC4, SHA-1/SHA-256/MD5/HMAC/PRNG, TRNG, PKA, and Secure Key Manager).

Even with the above assumption, how to provision the isolated secure environment in mobile devices is still a challenging task. In this thesis, three approaches are proposed in

the context of ARM virtualization environment, i.e., *Linux Container (LXC)-based Approach*, *Virtual Machine (VM)-based Approach*, and *Firmware TPM* (which is based on TrustZone Virtualization). *LXC-based approach* uses a LXC to protect the software-based TPM emulator, and *VM-based approach* uses a VM; these two approaches were implemented and evaluated. *Firmware TPM* refers to the alternate software implementation of the TPM specification in the context of a Trusted Execution Environment.

3.1. Linux Container-based Approach

Containers [7] are a lightweight virtualization technology. They are more akin to an enhanced *chroot* than to full virtualization like QEMU or VMWare, both because they do not emulate hardware and because containers share the same OS as the host. Therefore, containers are better compared to Solaris zones or BSD jails. *Linux-vserver* and *OpenVZ* are two pre-existing, independently developed implementations of containers-like functionality for Linux. In fact, containers came about as a result of the work to upstream the *vserver* and *OpenVZ* functionality. Some *vserver* and *OpenVZ* functionality is still missing in containers, however containers can boot many Linux distributions and have the advantage that they can be used with an unmodified upstream kernel.

Linux Containers (LXCs) [8] take a completely different approach than system virtualization technologies such as KVM and XEN, which started by booting separate virtual systems on emulated hardware and then attempted to lower their overhead via para-virtualization and related mechanisms. Instead of retrofitting efficiency onto full isolation, LXC started out with an efficient mechanism (existing Linux process management) and added isolation, resulting in a system virtualization mechanism as scalable and portable as *chroot*, capable of simultaneously supporting thousands of emulated systems on a single server while also providing lightweight virtualization options to routers and smart phones.

LXC implement:

1) Resource management via “process control groups”.

2) Resource isolation via namespaces, i.e., new flags to the clone(2) system call (capable of create several types of new namespaces for things like PIDs and network routing).

3.1.1. cgroups

cgroups (control groups) [9] is a Linux kernel feature that allows you to allocate resources—such as CPU time, system memory, network bandwidth, or combinations of these resources—among user-defined groups of tasks (processes) running on a system. You can monitor the *cgroups* you configure, deny *cgroups* access to certain resources, and even reconfigure your *cgroups* dynamically on a running system.

By using *cgroups*, fine-grained control over allocating, prioritizing, denying, managing, and monitoring system resources can be gained. Hardware resources can be smartly divided up among tasks and users, increasing overall efficiency.

For brevity of this section, the detailed introduction to *cgroups* is put into Appendices (8.1 Introduction to Linux Control Groups)

3.1.2. namespaces

Currently, Linux implements six different types of namespaces [10]. The purpose of each namespace is to wrap a particular global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. One of the overall goals of namespaces is to support the implementation of containers, a tool for lightweight virtualization.

For brevity of this section, the detailed introduction to *namespaces* is put into Appendices (8.2 Introduction to Linux Namespaces).

In essence, a LXC is a group of processes running on a system. This group of processes have the illusion that they are the only processes on the system, i.e., they form an isolated environment equivalent of a normal Linux system.

The rational of LXC-based approach is as follows: on top of a Linux kernel that has *cgroups* and *namespaces* enabled, three containers are created, i.e., *Management Container*,

TPM Emulator Container, and *User OS Container* (Fig. 3).

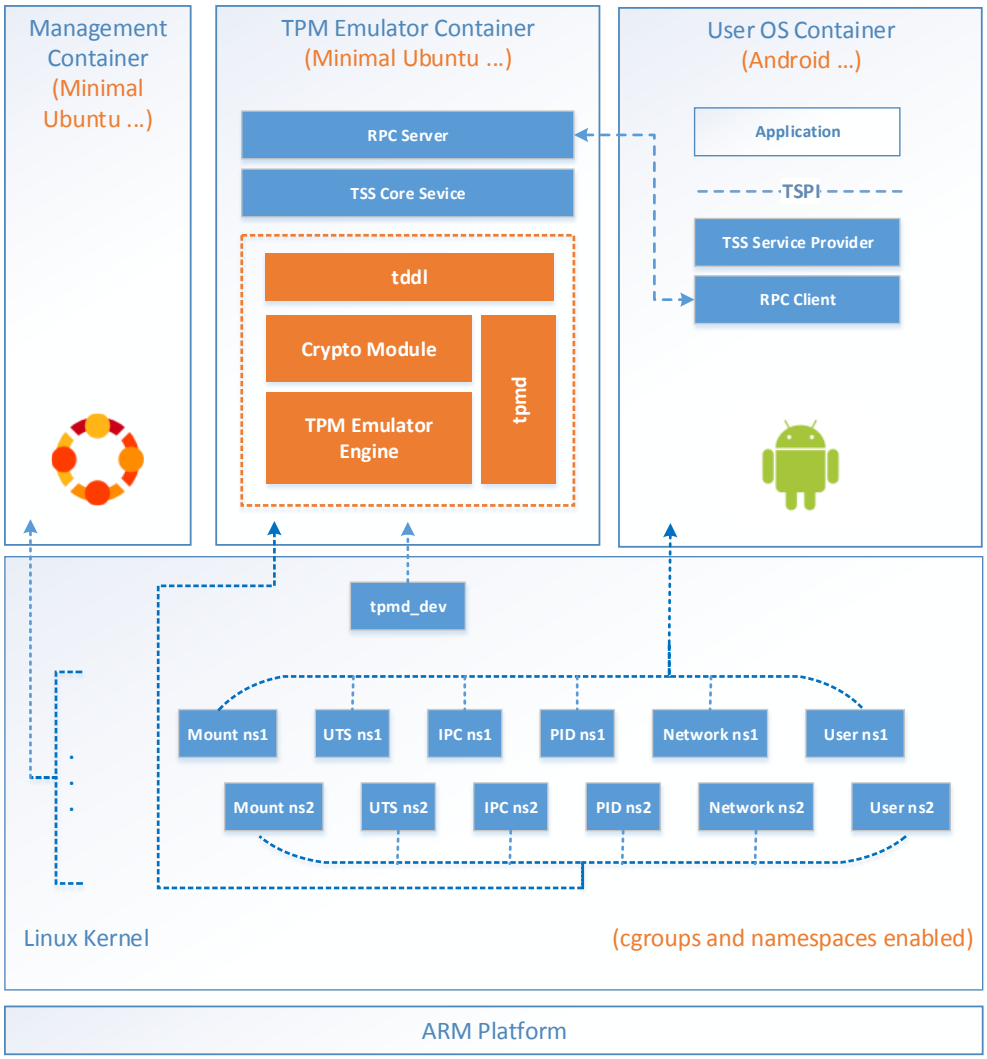


Fig. 3. Linux Container-based Approach

The *Management Container* is where a minimal system that provides the running environment for LXCs resides. It is responsible for managing other containers on the host. For example, this system can be a minimal Ubuntu system, or a minimal BusyBox-based system.

The *TPM Emulator* Container is where a minimal system that provides the running environment for the TPM Emulator resides. For example, this system can still be a minimal Ubuntu system, or a minimal BusyBox-based system. In addition, some parts of the TSS are installed in this container, i.e., the *TSS Core Service* and *TSS RPC Server*.

The *User OS Container* is where a normal device user's OS resides; the OS can be any Linux kernel-based OS for smart devices, including Android, Ubuntu for smartphones and tablets. In addition, some parts of the TSS are installed in this container, i.e., the *TSS RPC Client* and *TSS Service Provider*.

Note that there only exists one Linux kernel, which is shared by multiple containers.

3.2. Virtual Machine-based Approach

As the complexity of software increases, the requirement of multiple software environments to be available on the same physical processor increases simultaneously. Software applications that require separation for reasons of isolation, robustness or differing real-time characteristics need a virtual processor exhibiting the required functionality.

Providing virtual processors in an energy-efficient manner requires a combination of hardware acceleration and efficient software hypervisors. The ARM Architecture Virtualization Extension (VE) [11] standardizes the architecture for implementation of the hardware acceleration in ARM application processor cores, while high performance hypervisors from the world's leading virtualization companies provide the software component upon which to build effective software environments (i.e., VMs).

As VE is a recent ARM architecture extension, most ARM platforms are not VE-enabled. However, for non-VE enabled ARM platforms, mainstream Virtual Machine Monitors (VMMs) like XEN, KVM already have patches that making them run on non-VE enabled ARM platforms using para-virtualization. ARMvisor[12] is a para-virtualized VM, and it is based on KVM. The XEN ARM PV [13] project, led by Samsung, maintains an ARM variant of the XEN hypervisor in a codeline that is separate from the upstream XEN hypervisor project; it delivers and maintains XEN support for a range of ARM processors (ARMv5 – v7) for mobile devices, using XEN para-virtualization.

For non-VE enabled ARM platform, the deployment of ARMvisor and XEN ARM requires patches on the guest OS, thus not practical and LXC-based approach is recommended.

For VE-enabled ARM platforms, the guest OS does not require patches and thus the

deployment is much easier; the VE feature also makes the execution of a VM more efficient. KVM ARM [14] and XEN ARM PVH [15] already support VE-enabled ARM platforms, though not as mature as for x86 platforms. Therefore, for VE-enabled ARM platforms, VM-based approach is a good choice due to both the stronger isolation of a VM than a *Container* and the efficient execution of the VM.

For VM-based approach in this thesis, ARM VE-enabled platforms are focused on. The rationale of this approach is as follows: on top of a VMM, three VMs are created, i.e., *Management VM*, *TPM Emulator VM*, and *User VM* (Fig. 4).

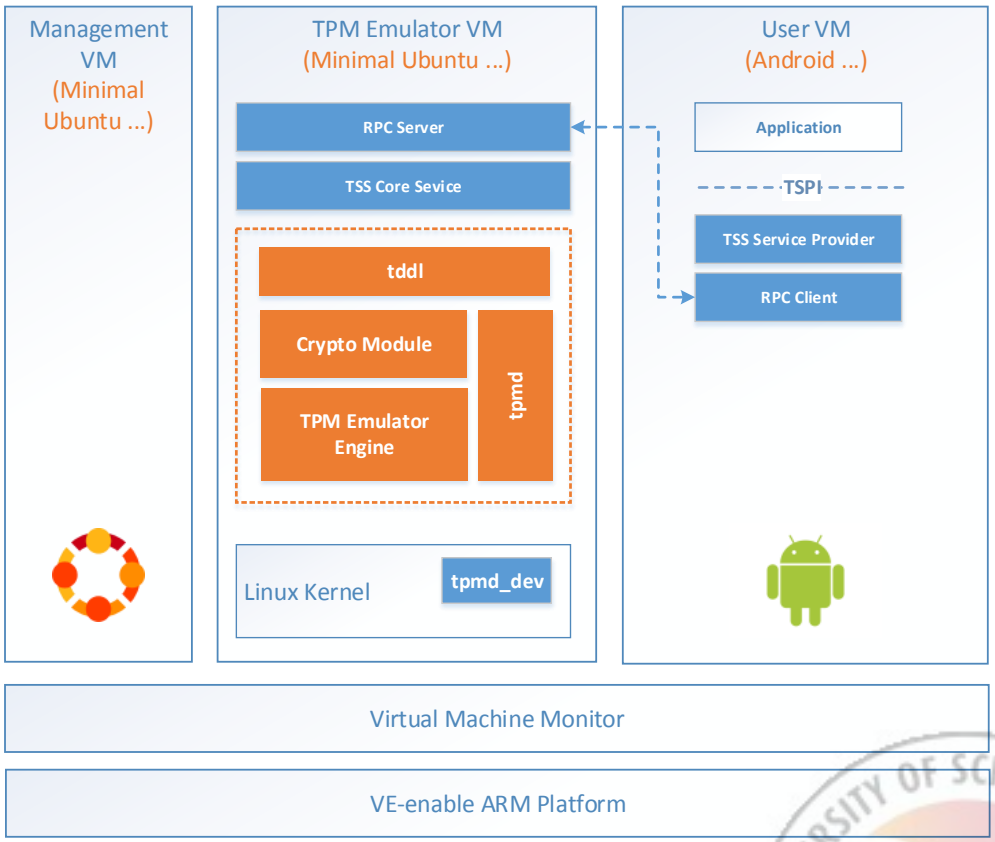


Fig. 4. Virtual Machine-based Approach

The *Management VM* is a VM inside which runs a minimal OS that manages other VMs on the host. For example, this OS can be a minimal Ubuntu system, or a minimal BusyBox-based system.

The *TPM Emulator VM* is a VM inside which runs a minimal OS that provides the running environment for the TPM emulator. For example, this OS can still be a minimal Ubuntu system, or a minimal BusyBox-based system. In addition, some parts of the TSS are installed in this VM, i.e., the *TSS Core Service* and *TSS RPC Server*.

The *User VM* is where a normal device user's OS resides; the OS can be any Linux kernel-based OSs for smart devices, including Android, Ubuntu for smartphones and tablets. In addition, some parts of the TSS are installed in this VM, i.e., the *TSS RPC Client* and *TSS Service Provider*.

3.3. Firmware TPM

GlobalPlatform [16] defined TEE specification. A TEE [17] is a secure area that resides in the application processor of an electronic device. To help visualize, think of a TEE as somewhat like a bank vault. A strong door protects the vault itself (hardware isolation) and within the vault, safety deposit boxes with individual locks and keys (software and cryptographic isolation) provide further protection.

Separated by hardware from the main OS, a TEE ensures the secure storage and processing of sensitive data and trusted applications. It protects the integrity and confidentiality of key resources, such as the user interface and service provider assets. A TEE manages and executes trusted applications built in by device makers as well as trusted applications installed as people demand them. Trusted applications running in a TEE have access to the full power of a device's main processor and memory, while hardware isolation protects these from user installed apps running in a main operating system. Software and cryptographic isolation inside the TEE protect the trusted applications contained within from each other.

Device and chip makers use TEEs to build platforms that have trust built in from the start, while service and content providers rely on integral trust to start launching innovative services and new business opportunities.

ARM TrustZone [18] technology provides the infrastructure foundation that allows a SoC designer to choose from a range of components to form a trusted environment that can fulfill specific functions. TrustZone API conforms to GlobalPlatform's TEE Client API, so a well-designed TrustZone-enabled SoC is the ideal foundation for a TEE. Current mobile devices are mainly ARM-based platforms, which typically have built-in TrustZone features, so it is possible to build a TEE in current mobile devices.

TPM MOBILE specification [19] explicitly supports its alternative implementations including as software. Combined with TEE, it is possible to implement the TPM in firmware, i.e., TPM MOBILE instance is implemented as a *Trusted Application* in the TEE. The reference architecture in TPM MOBILE 1.0 takes into account the support of several parallel TPM MOBILE instances in the same devices that can be used to secure different stakeholders in the device. Because a TEE is a hardware-isolated secure environment, it provides stronger protection for TPM MOBILE instances than LXC- and VM-based approaches.

The rationale of *Firmware TPM* is as follows: in a TrustZone-enabled ARM platform, some peripherals from the application processor (SoC) and the platform form the TEE (also refer to as secure world), and the remaining peripherals form the normal world for the device user's OS (Fig. 5).

Within the secure world, the *TrustZone Monitor* is responsible for the switching of the two worlds (secure world vs. normal world). The *TEE Kernel* schedules tasks on the whole system and exposes its services via TEE Internal API as defined by GlobalPlatform to TEE applications. *TPM MOBILE Instance* is a TEE application, whose functions are exposed to TSS's *TDDL*. On top of *TDDL* is TSS Core Service, which exposes all of TPM MOBILE Instance's capabilities and some of its own as defined by the TSS Specification.

In the normal world, device user's OS is running, such as Android, Ubuntu for smart phones and tablets. Applications use TPM's capabilities through the *TSPI*, which is exposed by *TSS Service Provider* as per the TSS Specification. The *TSS Service Provider* communicates with the secure world through the TEE Client API as defined by GlobalPlatform, where the *TSS Service Provider* passes the corresponding parameters to the *TSS Core Service* through a world switch and vice versa, the *TSS Core Service* passes back the result to *TSS Service Provider* via another world switch.

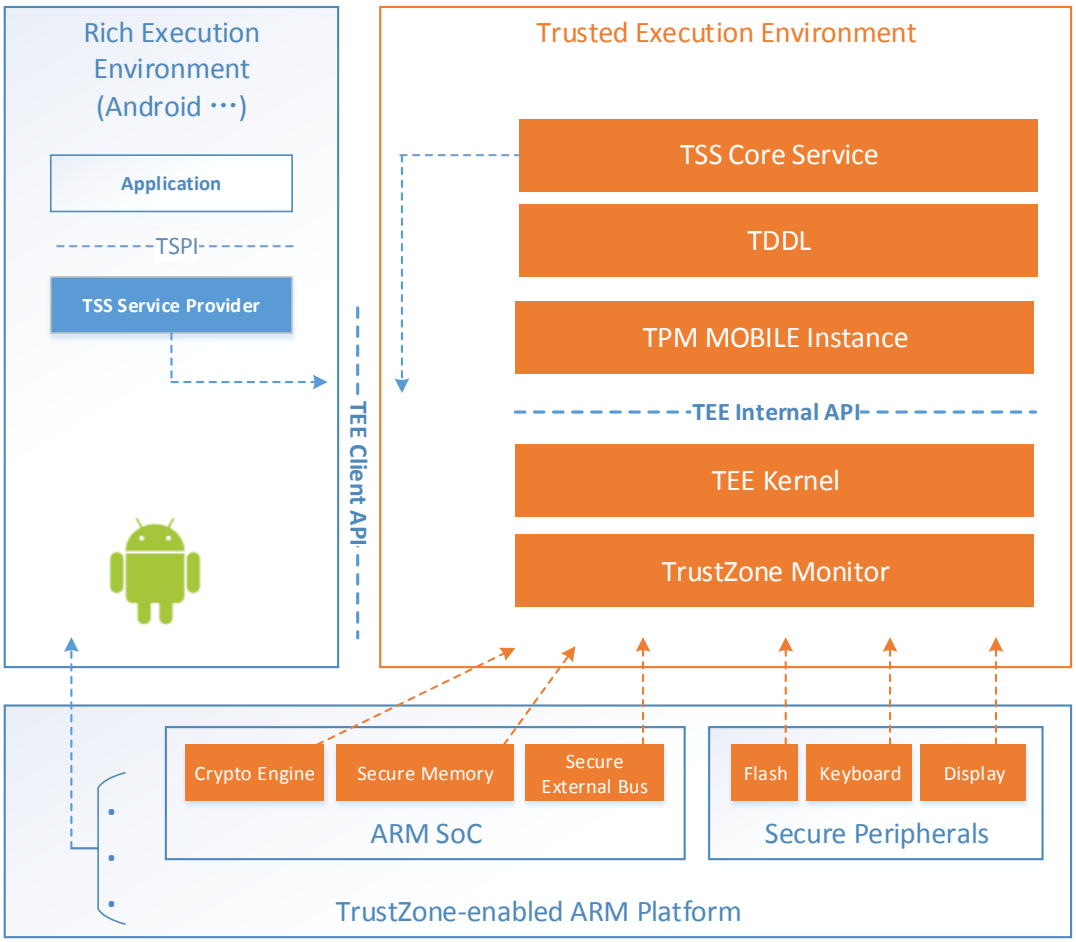


Fig. 5. Firmware TPM

4. Implementations

Of the three proposed approaches in the context of ARM virtualization environment, i.e., *Linux Container (LXC)-based Approach*, *Virtual Machine (VM)-based Approach*, and *Firmware TPM* (based on TrustZone Virtualization), *LXC*- and *VM-based* approaches are implemented and evaluated. The proposed reference architecture for *Firmware TPM* is given in the thesis, but its implementation is left as future work in that this approach requires far more programming effort than a master thesis project (refer to Section 6 Future Work for more details).

In addition, the implemented approaches also assume that the *roots of trust* of the TPM emulator are securely handled. As described in Section 3, this task can be aided by the device ROM and security subsystem of an ARM SoC.

4.1. Linux Container-based Approach

The implementation was done on an Arndale board, which runs a Cortex-A15 Application Processor (Samsung Exynos 5250). As this approach does not require the ARM VE feature, so it is applicable to any ARM platforms capable of running a Linux kernel.

As can be seen from Fig. 6, the board runs a 3.8 Linux kernel with *cgroups* and *namespaces* enabled. On top of the Linux kernel, two containers are running, the *TPM Emulator Container* and the *User OS Container*. The *TPM Emulator Container* is also the *Management Container* as specified in Fig. 4.

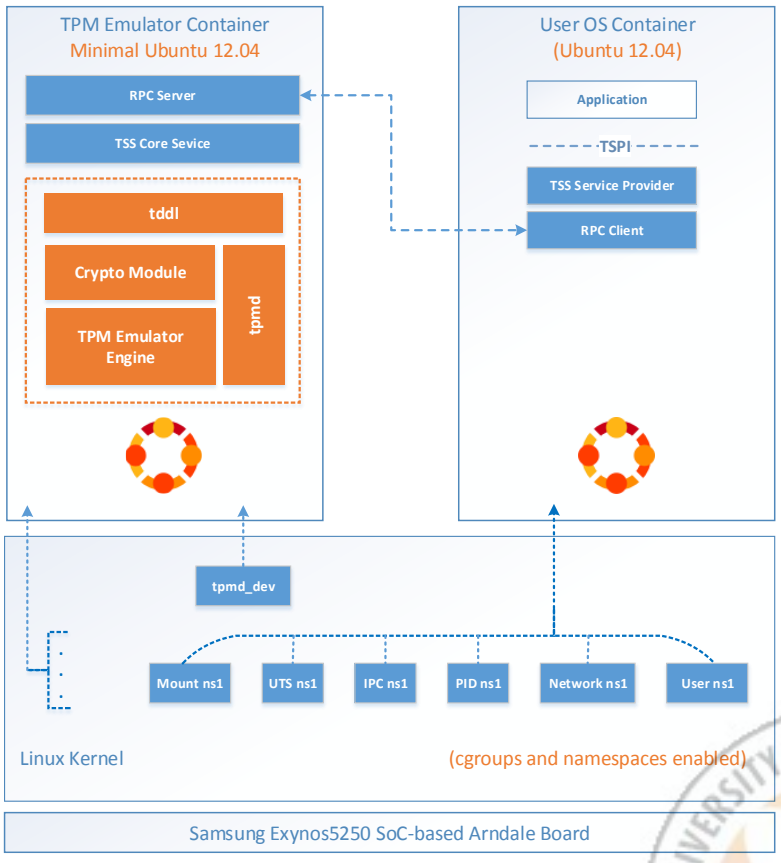


Fig. 6. Implementation of Linux Container-based Approach

The *TPM Emulator Container* runs a minimal Ubuntu 12.04 that provides the running environment for LXC and TPM Emulator. In addition, some parts of the TSS are installed in this container, i.e., the *TSS Core Service* and *TSS RPC Server*.

The *User OS Container* runs an Ubuntu 12.04 system. In addition, some parts of the TSS are installed in this container, i.e., the *TSS RPC Client* and *TSS Service Provider*. As the TPM emulator sometimes need to access device */dev/tpm*, so the container should be given permissions to access the kernel module *tpmd_dev*.

4.2. Virtual Machine-based Approach

Kernel-based Virtual Machine (KVM) [20, 21] is a full virtualization solution for Linux on platforms containing virtualization extensions (such as Intel VT, AMD-V, and ARM VE); it is a Linux kernel module that allows a user space program to utilize the hardware virtualization features of various processors. Today, it supports recent Intel and AMD processors (x86 and x86_64), PPC 440, PPC 970, S/390 and recent ARM processors (Cortex-A15, Cortex-A50). QEMU can make use of KVM when running a target architecture that is the same as the host architecture. For instance, when running *qemu-system-x86* on an x86 compatible processor, you can take advantage of the KVM acceleration – giving you benefit for your host and your guest system.

Using KVM, one can run multiple VMs running unmodified Linux or Windows images. Each VM has private virtualized hardware: a network card, disk, graphics adapter, etc.

The implementation of VM-based approach was also done on the Arndale board, which is VE-enabled. Because the implementation requires ARM VE feature of the platform processor, this approach is only applicable to VE-enabled ARM platforms.

As can be seen from Fig. 7, the board runs a 3.8 Linux kernel with the KVM kernel module loaded. Using KVM acceleration, *qemu-system-arm*, as a virtualizer, can run any unmodified ARM Linux or Windows image on each of its instance. On top of a *qemu-system-arm* instance, a VM of *RealView Versatile Express* model is emulated.

The *Host System* is both the *Management VM* and the *TPM Emulator VM*, which are specified in Fig. 5; it is a minimal Ubuntu 12.04 that provides the running environment for LXC and TPM Emulator. In addition, some parts of the TSS are installed in the host system, i.e., *TSS Core Service* and *TSS RPC Server*.

The RealView Versatile Express VM runs an Ubuntu 12.04 system, inside which some parts of the TSS are installed, i.e., *TSS RPC Client* and *TSS Service Provider*.

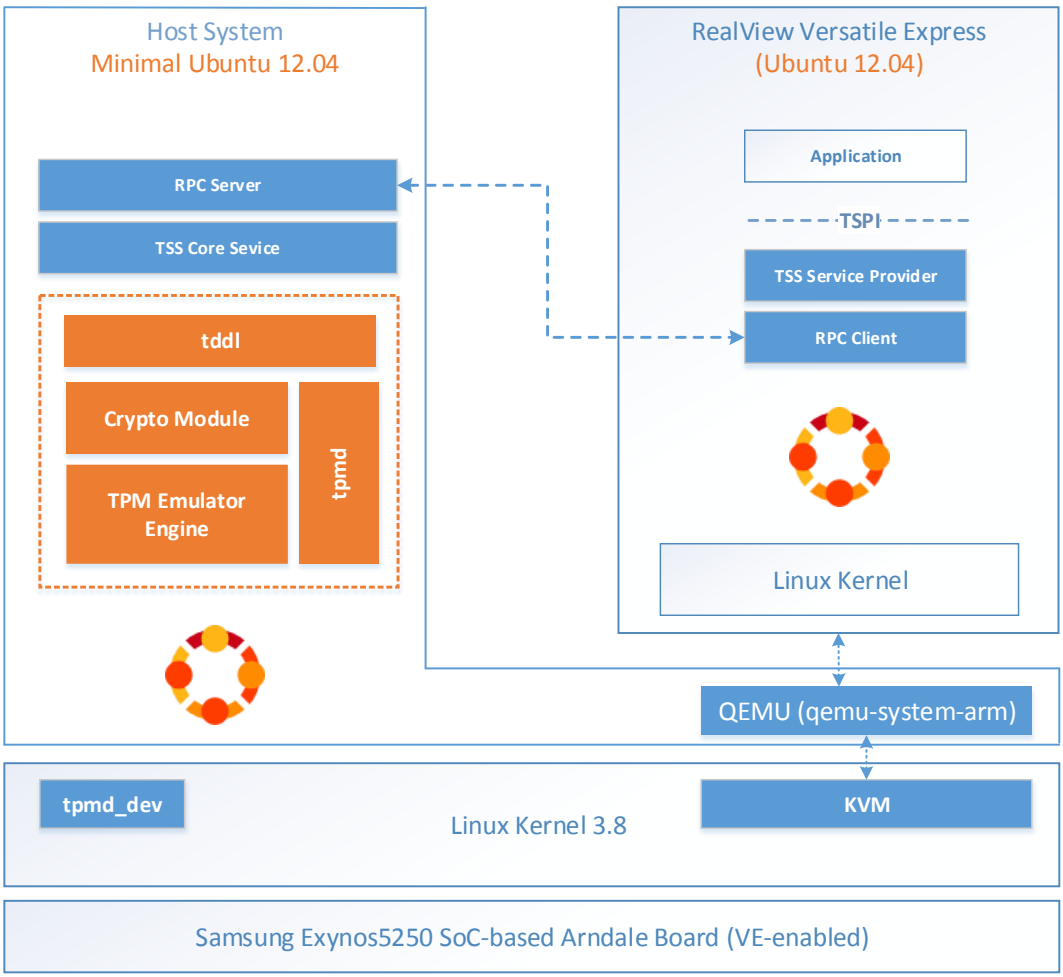


Fig. 7. Implementation of Virtual Machine-based Approach

5. Evaluation

5.1. Capability Evaluation of Software-based TPM Emulator

To evaluate to what extent the software-based TPM emulator provides capabilities of a hardware TPM, we just need to exercise the same test suite that contains the whole set of TPM commands against TPM emulator and hardware TPM respectively, and then compare the results.

Commands to the TPM go through the TSS. There are two types of commands that go to the TSS: those that are known to be secure, and those that are not needed to be secure. Commands that need not to be secure are not usually implemented in the TPM, i.e., implemented in the TSS, such as bind and verify signature; commands that must be secure are implemented in hardware.

As the TSPI of TSS exposes all the TPM's capabilities and some of its own, we can use the TSPI to exercise the TPM commands. IBM TrouSerS [22] is an open-source TSS and provides the TSS API test suite, which will be used in the evaluation.

The evaluation exercised all the commands implemented in the TPM. For commands implemented in the TSS, they are not tested as they are TSS implementation dependent. Table 1 provides the evaluation result, and evaluation result of individual command is attached in Appendices (8.3 Capability Evaluation Result of Software-based TPM Emulator)

In the total 168 exercised TPM commands, the Infineon hardware TPM (v1.2) passed 162 commands (plus 6 obsolete or not implemented commands), while software-based TPM emulator (LXC- and VM-based approaches) passed all the tests, i.e., the software-based

TPM emulator provides the same capabilities of a hardware TPM *in terms of TPM commands*.

Table 1. Evaluation Result of Commands Implemented in the TPM

	Infineon TPM 1.2	**TPM Emulator (LXC)**	**TPM Emulator (VM)**
Total Commands	168		
Passed	162 (+ 6 obsolete or not implemented commands)	168	168
Failed	0	0	0
Segfaulted	0	0	0

5.2. Overhead Evaluation of Software-based TPM Emulator

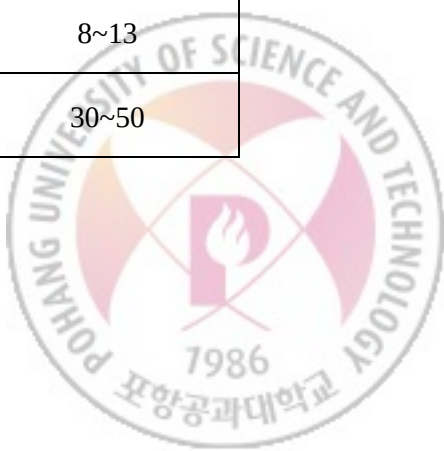
To evaluate the memory overhead of the software-based TPM emulator, two test cases are used. One test case is when the TPM emulator is running without any commands issued, the other test case is when the TPM emulator is running with one command issued per second.

To evaluate the CPU overhead of the software-based TPM emulator, three test cases are used. The first test case is when the TPM emulator is running without any commands issued, the second test case is when the TPM emulator is running with one command issued every 10 seconds, and the third test case is one command issued per second.

Table 2 shows the evaluation result. In terms of memory overhead, *LXC-based approach* is quite acceptable; As the *VM-based approach* is mainly targeted at VE-enabled ARM platforms, which typically have more than 2 GB memory, thus the overhead is also acceptable.

Table 2. Evaluation Result of Overhead of Software-based TPM Emulator

		TPM Emulator (LXC)	**TPM Emulator (VM)**
Memory (MB)	No command issued	21	66
	One command per second	23	71
CPU (%)	No command issued	1~2	1~2
	1 command every 10 seconds	5~6	8~13
	One command per second	20~30	30~50



In terms of CPU overhead, when the command issuing frequency is less than 1 command every 10 seconds, the CPU overhead of both approaches is quite acceptable; when the command issuing frequency is above 1 command per second, the CPU overhead is not acceptable. However, the TPM command issuing frequency is seldom exceeds 1 command per second and is usually quite lower than 1 command every 10 seconds. Therefore, generally speaking, the CPU overhead is acceptable.

5.3. Execution Speed Evaluation of Software-based TPM Emulator

Because we cannot install a hardware TPM on the Arndale development board, in this evaluation we cannot compare the execution time of the software based-TPM emulator against a hardware TPM.

To compare TPM command execution speed of software-based TPM emulator under *Linux Container*- and *Virtual Machine*-based approaches, three typical TPM commands were chosen. The evaluation result is shown in Table 3.

Table 3 Evaluation Result of Execution Speed (tick) of Software-based TPM Emulator

TPM Command	**TPM Emulator (LXC)**	**TPM Emulator (VM)**
GetAuditDigest	110	134
PcrExtend	5	7
GetRandom	5	6

In the Linux kernel we used, the system has 100 ticks per second. The execution speed is measured in terms of ticks. From the evaluation result, we can see that *Linux Container*-based approach is faster than *Virtual Machine*-based approach.

6. Future Work

6.1. Optimizing Current Work

BusyBox [24] combines tiny versions of many common UNIX utilities into a single small executable. It provides replacements for most of the utilities you usually find in GNU fileutils, shellutils, etc. The utilities in BusyBox generally have fewer options than their full-featured GNU cousins; however, the options that are included provide the expected functionality and behave very much like their GNU counterparts. BusyBox provides a fairly complete environment for any small or embedded system. BusyBox has been written with size-optimization and limited resources in mind. It is also extremely modular so you can easily include or exclude commands (or features) at compile time. This makes it easy to customize your embedded systems. To create a working system, just add some device nodes in /dev, a few configuration files in /etc, and a Linux kernel.

The implementation of *Linux Container-based approach* uses a minimal Ubuntu as the *Management Container* and *TPM Emulator Container*. As the responsibility of *Management Container* is to support the running of Linux containers, and TPM Emulator Container the running of VMs, we can continue to reduce the overhead by using a BusyBox-based system.

The above principle can be applied to Virtual Machine-based approach equally.

Using a BusyBox-based system instead of the minimal Ubuntu system, the overhead can be reduced significantly. However, determining and adding the dependencies of the necessary environment (that supports the running of LXC and TPM emulator) other than those provided in BusyBox can be quite troublesome, though not a difficult task.

6.2. Future Implementation Considerations of Firmware TPM

Firmware TPM refers to the alternate software implementation of the TPM specification in the context of a Trusted Execution Environment; the proposed reference architecture for *Firmware TPM* is given Fig. 3, but its implementation is left as future work in that this approach requires far more programming effort than a master's thesis project.

To implement *Firmware TPM*, the most important thing is to handle the *roots of trust* securely. With the above assumption, there are still many challenges. The following components must be well implemented (See Fig. 3 for more details) and seamlessly integrated.

1) TrustZone Monitor

TrustZone Monitor [25] is the software that manages the switches between the *secure* and *non-secure* processor states. In most designs its functionality will be similar to a traditional OS context switch, ensuring that state of the world that the processor is leaving is safely saved, and the state of the world the processor is switching to is correctly restored.

Currently, there are two open-source *TrustZone Monitors* that can be referenced to, i.e., Open Virtualization TrustZone Monitor [26] and SafeG [27].

2) TEE Kernel

The TEE kernel is a secure kernel running inside the TEE. It schedules TEE tasks, i.e., TEE applications. It also provides drivers for the Rich OS ("normal world") to communicate with the secure world ("secure world").

Currently, there are three open-source TEE kernels that can be referenced to, i.e., Open Virtualization TEE kernel [26] and TEE kernels for SafeG [27] (TOPPERS ASP,

TOPPERS FMP).

3) TPM MOBILE Instance

This is the software component that implements the TPM commands as specified by the TPM specification. In essence, this is a TEE application, which should conform to the TEE Internal API specification. The implementation can refer to *TPM emulator* project with the TEE Internal API in mind.

4) TSS

This is the software component that exposes all of *Firmware TPM*'s capabilities and some of its own capabilities. Some parts of TSS should be run in TEE, i.e., *TDDL* and *TSS Core Service*; other parts are run in the normal word, i.e., *TSS Service Provider*.

However, a communication protocol between *TSS Service Provider* and *TSS Core Service* should be well established in accordance to TEE Client API specification. In view of the large amount of functions exposed by the *TSS Core Service*, this would be a very difficult task.

In addition, the testing to verify the correct implementation of all the above components would also be a very difficult task.

7. Conclusion

As mobile technology matures, mobile devices (principally smart phones and tablets) are increasingly being used in both personal and corporate environments. While mobile devices bring great convenience to us, security problems also ensue. Current mobile devices lack the hardware-based root of trust features (e.g., Trusted Platform Module, or TPM) that are increasingly built into laptops, PCs and other types of hosts. Unfortunately, mobile devices are constrained in space, cost and power dimensions that make the use of a discrete TPM difficult.

Since a software-based TPM emulator can provide the same capabilities of a hardware TPM in terms of *TPM Commands*, thus, if 1) *the roots of trust of the TPM emulator can be securely handled*, 2) *an isolated secure environment can be provisioned in the mobile device, be it software environment or hardware environment*, without the presence of a hardware TPM, the approximate same capabilities of a hardware TPM can be obtained by running a software-based TPM emulator in the said secure environment. In this thesis, we assume the *roots of trust* of a software-based TPM emulator are securely handled. However, even with the above assumption, how to provision the isolated secure environment is still a challenging task.

In this thesis, three approaches are proposed in the context of ARM virtualization environment, i.e., *Linux Container (LXC)-based Approach*, *Virtual Machine (VM)-based Approach*, and *Firmware TPM* (which is based on TrustZone Virtualization). *LXC-based approach* uses a LXC to protect the software-based TPM emulator, and *VM-based approach* uses a VM; these two approaches were implemented and evaluated. *Firmware TPM* refers to the alternate software implementation of the TPM specification in the context of a Trusted Execution Environment; the proposed

reference architecture for *Firmware TPM* is given in the thesis, but its implementation is left as future work in that this approach requires far more programming effort than a master's thesis project.

The evaluation result showed that the software-based TPM emulator can provide the same capabilities of a hardware TPM provided that 1) the *roots of trust* of the emulator are securely handled, 2) the proposed secure environments (Linux Container or VM) are secure enough. Though the addition of a software-based TPM emulator on a mobile device incurs some overhead on the system, the overhead is acceptable for modern mobile devices, which typically shipped with more than 1GB memory and 1GHZ or faster CPUs.

Limitation of Current Work Having stated in the beginning of *Section 3*, current work assumes that the *roots of trust* are securely handled. This limitation cannot be easily handled, which can be aided be device ROM or security subsystem of an application processor. As this may require some proprietary firmware of CPU or device manufacturers, it is not a trivial job.

Making A Decision When it comes to which approach should be adopted between *Linux Container*- and *Virtual Machine*-based approaches, generally speaking, for VE-enabled devices, like Chrome book, both approaches are acceptable, while for non-VE enabled devices, *Linux Container*-based approach is recommended.

8. Appendices

8.1. Introduction to Linux Control Groups

control groups (*cgroups*) [9] is a Linux kernel feature that allows you to allocate resources—such as CPU time, system memory, network bandwidth, or combinations of these resources—among user-defined groups of tasks (processes) running on a system. You can monitor the *cgroups* you configure, deny *cgroups* access to certain resources, and even reconfigure your *cgroups* dynamically on a running system.

By using *cgroups*, fine-grained control over allocating, prioritizing, denying, managing, and monitoring system resources can be gained. Hardware resources can be smartly divided up among tasks and users, increasing overall efficiency.

8.1.1. How Control Groups Are Organized

cgroups [9] are organized hierarchically, like processes, and child groups inherit some of the attributes of their parents. However, there are differences between the two models.

The Linux Process Model. All processes on a Linux system are child processes of a common parent: the *init* process, which is executed by the kernel at boot time and starts other processes. Because all processes descend from a single parent, the Linux process model is a single hierarchy, or tree.

The cgroups Model. *cgroups* are similar to processes in that: they are hierarchical; child *cgroups* inherit certain attributes from their parent *cgroup*. The fundamental difference is that many different hierarchies of *cgroups* can exist simultaneously on a system. If the Linux process model is a single tree of processes, then the *cgroup* model is one or more separate, unconnected trees of tasks (i.e., processes).

Multiple separate hierarchies of *cgroups* are necessary because each hierarchy is attached to one or more subsystems. A subsystem [23] represents a single resource, such as CPU time or memory, which is typically a "resource controller" that schedules a resource or applies per-cgroup limits, but it may be anything that want to act on a group of processes, e.g., a virtualization subsystem.

A hierarchy [23] is a set of *cgroups* arranged in a tree, such that every task in the system is in exactly one of the *cgroups* in the hierarchy, and a set of subsystems; each subsystem has system-specific state attached to each cgroup in the hierarchy. Each hierarchy has an instance of the *cgroup* virtual filesystem associated with it.

Also note that at any one time there may be multiple active hierarchies of task *cgroups*. Each hierarchy is a partition of all tasks in the system.

8.1.2. Relationships between Subsystems, Hierarchies, Control Groups and Tasks

The relationships between subsystems, hierarchies, control groups and tasks are governed by the following rules.

Rule 1. A single hierarchy can have one or more subsystems attached to it.

As a consequence, the ***cpu*** and ***memory*** subsystems (or any number of subsystems) can be attached to a single hierarchy, as long as each one is not attached to any other hierarchy which has any other subsystems attached to it already.

Rule 2. Any single subsystem (such as **cpu**) cannot be attached to more than one hierarchy if one of those hierarchies already has a different subsystem attached to it already.

As a consequence, the **cpu** subsystem can never be attached to two different hierarchies if one of these of hierarchies already has the memory subsystem attached to it. However, a

single subsystem can be attached to two hierarchies if both of those hierarchies have only that subsystem attached.

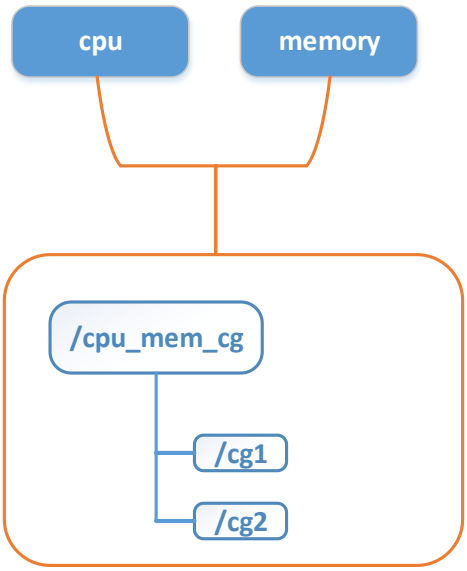
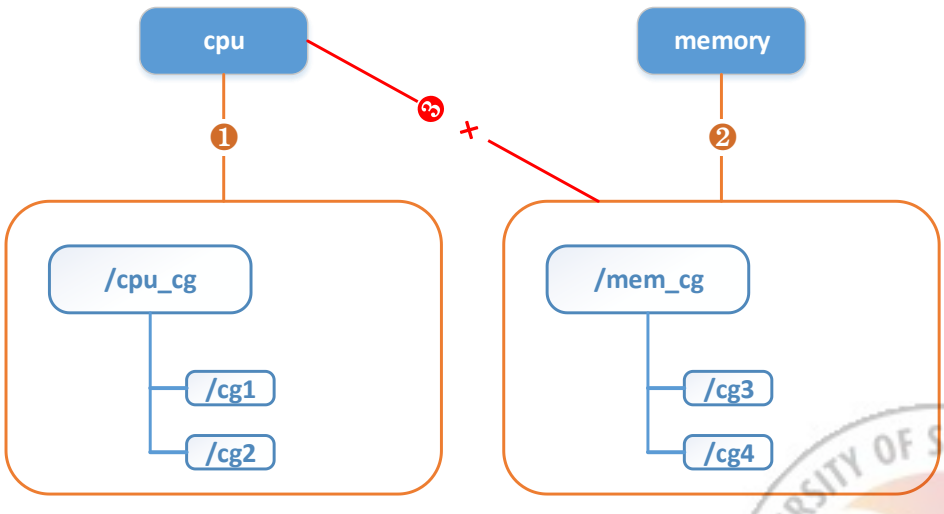


Fig. 8. cgroups Rule 1



* The numbered bullets represent a time sequence in which the subsystems are attached.

Fig. 9. cgroups Rule 2

Rule 3. Each time a new hierarchy is created on the system, all tasks on the system are initially members of the default cgroup of that hierarchy, which is known as the *root cgroup*. For any singe hierarchy you create, each task on the system can be a member of exactly one cgroup in that hierarchy. A single task may be in multiple cgroups, as long as each of those cgroups is in a different hierarchy. As soon as a task becomes a member of a second cgroup in the same hierarchy, it is removed from the first group in that hierarchy. At no time is a task ever in two different cgroups in the same hierarchy.

As a consequence, if the **cpu** and **memory** subsystems are attached to a hierarchy named **cpu_mem_cg**, and the **net_cls** subsystem is attached to a hierarchy named **net**, then a running **httpd** process could be a member of anyone cgroup in **cpu_mem_cg**, and any one cgroup in **net**.

When the first hierarchy is created, every task on the system is a member of at least one cgroup: the root cgroup. When using cgroups, therefore, every system task is always in at least one cgroup.

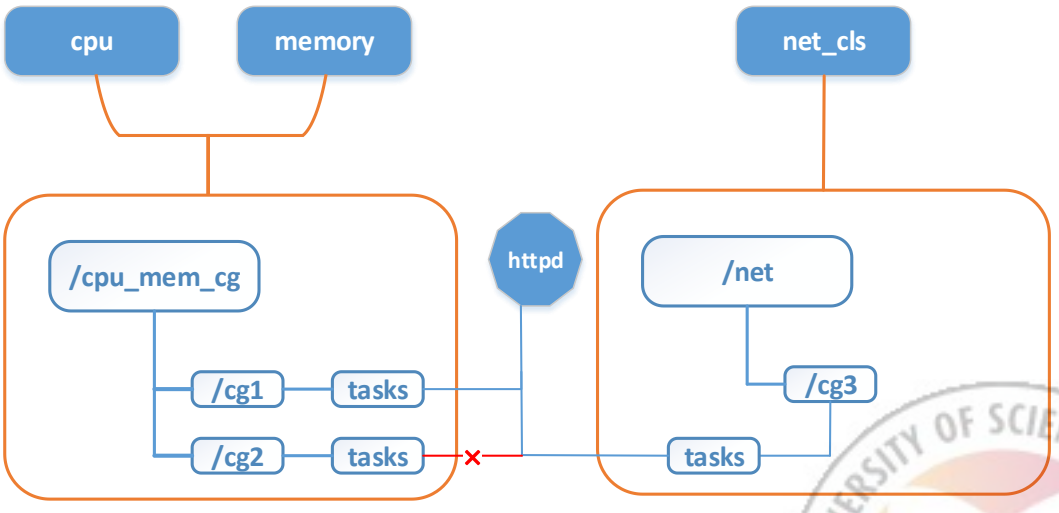
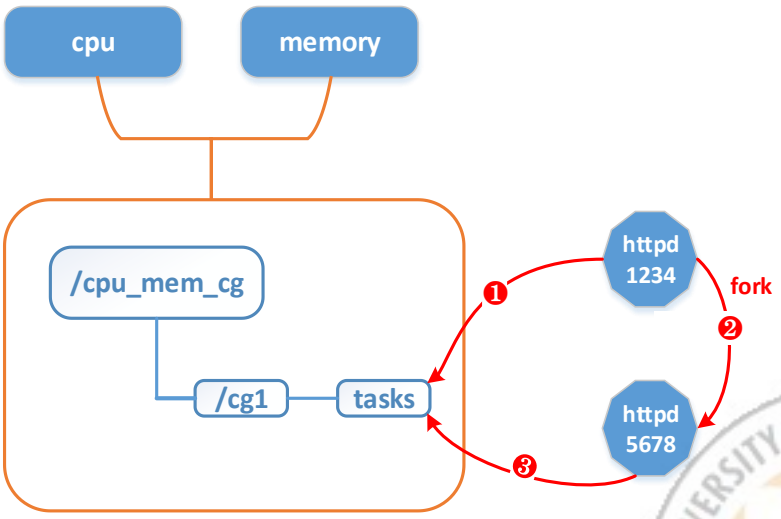


Fig. 10. cgroups Rule 3

Rule 4. Any process (task) on the system which forks itself creates a child process (task). A child task automatically inherits the cgroup membership of its parent but can be moved to different cgroups as needed. Once forked, the parent and child processes are completely independent.

As a consequence, consider the **httpd** task that is a member of the cgroup named **half_cpu_1gb_max** in the **cpu_and_mem** hierarchy, and a member of the cgroup **trans_rate_30** in the **net** hierarchy. When that **httpd** process forks itself, its child process automatically becomes a member of the **half_cpu_1gb_max** cgroup, and the **trans_rate_30 cgroup**. It inherits the exact same cgroups its parent task belongs to.

From that point on, the parent and child tasks are completely independent of each other: changing the cgroups that one task belongs to does not affect the other. Neither will changing cgroups of a parent task affect any of its grandchildren in any way. To summarize: any child task always initially inherit memberships to the exact same cgroups as their parent task, but those memberships can be changed or removed later.



* The number bullets represent a time sequence in which the task forks.

Fig. 11. cgroups Rule 4

8.2. Introduction to Linux Namespaces

Currently, Linux implements six different types of namespaces [10]. The purpose of each namespace is to wrap a particular global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. One of the overall goals of namespaces is to support the implementation of containers, a tool for lightweight virtualization (as well as for other purposes) that provides a group of processes with the illusion that they are the only processes on the system.

Mount namespaces isolate the set of filesystem mount points seen by a group of processes. Thus, processes in different mount namespaces can have different views of the filesystem hierarchy. With the addition of mount namespaces, the *mount()* and *umount()* system calls ceased operating on a global set of mount points visible to all processes on the system and instead performed operations that affected just the mount namespace associate with the calling process.

One use of mount namespaces is to create environments that are similar to *chroot* jails. However, by contrast with the use of the *chroot()* system call, mount namespaces are a more secure and flexible tool for this task. Other more sophisticated uses of mount namespaces are also possible. For example, separate mount namespaces can be set up in a master-slave relationship, so that the mount events are automatically propagated from one namespace to another; this allows, for example, an optical disk device that is mounted in one namespace to automatically appear in other namespaces.

UTS namespaces isolate two system identifiers—*nodename* and *domainname*—returned by *uname()* system call; the names are set using the *sethostname()* and *setdomainname()* system calls. In the context of containers, the UTS namespaces feature allows each container to have its own hostname and NIS domain name. This can be useful for

initialization and configuration scripts that tailor their actions based on these names. The term "UTS" derives from the name of the structure passed to the *uname()* system call: *struct utsname*. The name of that structure in turn derives from "UNIX Time-sharing System".

IPC namespaces isolate certain interprocess communication (IPC) resources, namely, System V IPC objects and (since Linux 2.6.30) POSIX message queues. The common characteristic of these IPC mechanisms is that IPC objects are identified by mechanisms other than filesystem pathnames. Each IPC namespace has its own set of System V IPC identifiers and its own POSIX message queue filesystem.

PID namespaces isolate the process ID number space. In other words, processes in different PID namespaces can have the same PID. One of the main benefits of PID namespaces is that containers can be migrated between hosts while keeping the same process IDs for the processes inside the container. PID namespaces also allow each container to have its own *init* (PID 1), the "ancestor of all processes" that manages various system initialization tasks and reaps orphaned child processes when they terminate.

From the point of view of a particular PID namespace instance, a process has two PIDs: the PID inside the namespace, and the PID outside the namespace on the host system. PID namespaces can be nested: a process will have one PID for each of the layers of the hierarchy starting from the PID namespace in which it resides through to the root PID namespace. A process can see (e.g., view via /proc/PID and send signals with *kill()*) only processes contained in its own PID namespace and the namespaces nested below that PID namespace.

Network namespaces provide isolation of the system resources associated with networking. Thus, each network namespace has its own network devices, IP addresses, IP routing tables, */proc/net* directory, port numbers, and so on.

Network namespaces make containers useful from a networking perspective: each container can have its own (virtual) network device and its own applications that bind to the per-namespace port number space; suitable routing rules in the host system can direct network packets to the network device associated with a specific container. Thus, for example, it is possible to have multiple containerized web servers on the same host system, with each server bound to port 80 in its (per-container) network namespace.

User namespaces isolate the user and group ID number spaces. In other words, a process's user and group IDs can be different inside and outside a user namespace. The most interesting case here is that a process can have a normal unprivileged user ID outside a user namespace while at the same time having a user ID of 0 inside the namespace. This means that the process has full root privileges for operations inside the user namespace, but is unprivileged for operations outside the namespace.

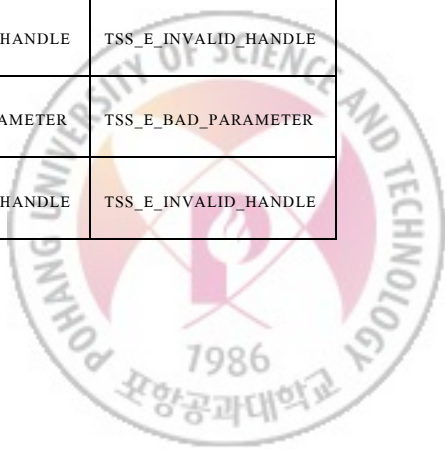
Starting Linux kernel 3.8, unprivileged processes can create user namespaces, which opens up a raft of interesting new possibilities for applications: since an otherwise unprivileged process can hold root privileges inside the user namespace, unprivileged applications now have access to functionality that was formerly limited to root.

8.3. Capability Evaluation Result of Software-based TPM Emulator

TPM Command	Infineon TPM 1.2	TPM Emulator (LXC)	TPM Emulator (VM)
Tspi_TPM_AuthorizeMigrationTicket01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_AuthorizeMigrationTicket02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_AuthorizeMigrationTicket03	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_AuthorizeMigrationTicket04	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_CheckMaintenancePubKey01	TPM_E_BAD_ORDINAL	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_ClearOwner02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_CollateIdentityRequest01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_CreateEndorsementKey01	TPM_E_DISABLED_CMD	TPM_E_DISABLED_CMD	TPM_E_DISABLED_CMD
Tspi_TPM_CreateEndorsementKey02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_CreateMaintenanceArchive01	TPM_E_BAD_ORDINAL	TPM_E_DISABLED_CMD	TPM_E_DISABLED_CMD
Tspi_TPM_CreateMaintenanceArchive02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Delegate_AddFamily01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_Delegate_AddFamily02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Delegate_AddFamily03	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Delegate_AddFamily04	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_Delegate_CacheOwnerDelegation01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS

Tspi_TPM_Delegate_CacheOwnerDelegation02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Delegate_CacheOwnerDelegation03	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Delegate_CacheOwnerDelegation04	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_Delegate_CacheOwnerDelegation05	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Delegate_CreateDelegation02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Delegate_CreateDelegation03	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Delegate_CreateDelegation04	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_Delegate_CreateDelegation05	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Delegate_CreateDelegation06	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Delegate_CreateDelegation07	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Delegate_CreateKeyDelegation01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_Delegate_CreateOwnerDelegation01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_Delegate_GetFamily01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_Delegate_GetFamily02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Delegate_GetFamily03	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_Delegate_GetFamily04	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_Delegate_GetFamily05	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Delegate_InvalidateFamily01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS

Tspi_TPM_Delegate_InvalidateFamily02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Delegate_InvalidateFamily03	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Delegate_InvalidateFamily04	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Delegate_ReadTables01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_Delegate_ReadTables02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Delegate_ReadTables03	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_Delegate_ReadTables04	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_Delegate_ReadTables05	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_Delegate_ReadTables06	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_Delegate_UpdateVerificationCount01	TPM_E_BADINDEX	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_Delegate_UpdateVerificationCount02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Delegate_UpdateVerificationCount03	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Delegate_VerifyDelegation01	TPM_E_FAMILYCOUNT	TPM_E_FAMILYCOUNT	TPM_E_FAMILYCOUNT
Tspi_TPM_Delegate_VerifyDelegation02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_DirRead01	TPM_E_BAD_ORDINAL	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_DirRead02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_DirRead03	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_DirRead04	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE



Tspi_TPM_DirWrite01	TPM_E_BAD_ORDINAL	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_DirWrite02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_DirWrite03	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_GetAuditDigest01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetAuditDigest02	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetAuditDigest03	TSS_SUCCESS	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_GetAuditDigest04	TSS_SUCCESS	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_GetAuditDigest05	TSS_SUCCESS	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_GetAuditDigest06	TSS_SUCCESS	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_GetAuditDigest07	TSS_SUCCESS	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_GetAuditDigest08	TSS_SUCCESS	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_GetAuditDigest09	TSS_SUCCESS	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_GetAuditDigest10	TSS_SUCCESS	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_GetAuditDigest11	TSS_SUCCESS	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_GetCapability01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability02	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_GetCapability03	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_GetCapability04	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS

Tspi_TPM_GetCapability05	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability06	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability07	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability08	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability09	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability10	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability11	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability12	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability13	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability14	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability15	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability16	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability17	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability18	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability19	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability20	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability21	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability22	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS

Tspi_TPM_GetCapability23	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability25	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability26	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability27	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetCapability29	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetEvent01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetEvent02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_GetEvent03	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_GetEvent04	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_GetEvent05	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_GetEventLog01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetEventLog02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_GetEventLog03	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_GetEvents01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetEvents02	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_GetEvents03	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_GetEvents04	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_GetEvents05	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER

Tspi_TPM_GetPubEndorsementKey01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetPubEndorsementKey02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_GetPubEndorsementKey03	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_GetPubEndorsementKey04	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_GetRandom01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetRandom02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_GetRandom03	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_GetStatus01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetStatus02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_GetStatus03	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_GetTestResults01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_GetTestResults02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_GetTestResults03	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_KeyControlOwner01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_KillMaintenanceFeature01	TPM_E_BAD_ORDINAL	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_KillMaintenanceFeature02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_LoadMaintenancePubKey01	TSS_E_BAD_PARAMETER	TPM_E_DISABLED_CMD	TPM_E_DISABLED_CMD
Tspi_TPM_LoadMaintenancePubKey02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE

Tspi_TPM_OwnerGetSRKPubKey01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_OwnerGetSRKPubKey02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_OwnerGetSRKPubKey03	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_OwnerGetSRKPubKey04	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_PcrExtend01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_PcrExtend02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_PcrExtend03	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_PcrExtend04	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_PcrRead01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_PcrRead02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_PcrRead03	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_PcrReset01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_PcrReset02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_PcrReset03	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Quote01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_Quote02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Quote03	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Quote04	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE

Tspi_TPM_Quote06	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_Quote2_01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_Quote2_02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Quote2_03	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Quote2_04	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_Quote2_05	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_Quote2_06	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_Quote2_07	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_ReadCounter01	TSS_E_NO_ACTIVE_COUNTER	TSS_E_NO_ACTIVE_COUNTER	TSS_E_NO_ACTIVE_COUNTER
Tspi_TPM_ReadCounter02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_ReadCounter03	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_ReadCurrentTicks01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_ReadCurrentTicks02	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_ReadCurrentTicks03	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_RevokeEndorsementKey02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_RevokeEndorsementKey03	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_RevokeEndorsementKey04	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER

Tspi_TPM_SelfTestFull01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_SelfTestFull02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_SetStatus01	TSS_E_BAD_PARAMETER	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_SetStatus02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_SetStatus03	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_StirRandom01	TSS_SUCCESS	TSS_SUCCESS	TSS_SUCCESS
Tspi_TPM_StirRandom02	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE
Tspi_TPM_StirRandom03	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER	TSS_E_BAD_PARAMETER
Tspi_TPM_TakeOwnership02	TPM_E_BAD_ORDINAL	TSS_E_INVALID_HANDLE	TSS_E_INVALID_HANDLE

Publications

1. Mingyuan Gao, Jaebok Shin, Wooram Park and Chanik Park, Using Software-based TPM Emulator in ARM-based Platforms, The 8th International Symposium on Embedded Technology, Daegu, Korea

REFERENCES

1. David Challener, Kent Yoder, et al., “A Practical Guide to Trusted Computing”, IBM Press, 2008.
2. Lily Chen, Joshua Franklin, and Andrew Regenscheid, "Guidelines on Hardware-Rooted Security in Mobile Devices (Draft)", NIST Special Publication 800-164 (Draft), October, 2012.
3. David Fisher et al., “Trust and Trusted Computing Platforms”. Carnegie Mellon University Software Engineering Institute, Paper 642, http://repository.cmu.edu/sei /642/.
4. Andrew Martin, “The ten-page introduction to Trusted Computing”. Oxford University Computing Laboratory, Software Engineering Group, CS-RR-08-11.
5. TPM Emulator Project, http://tpm-emulator.berlios.de/index.html.
6. Samsung, Samsung Exynos 5 Dual (Exynos 5250) User’s Manual, http://www. samsung.com/global/business/semiconductor/file/product/Exynos_5_Dual_User_ Manaul_Public_REV100-0.pdf.
7. Canonical, “Ubuntu Server Guide”, https://help.ubuntu.com/13.04/serverguide/ serverguide.pdf.
8. Linux Container Project, http://lxc.sourceforge.net/.
9. Martin Prpič, Rüdiger Landmann, et al., “Red Hat Enterprise Linux 6 Resource Management Guide”, https://access.redhat.com/site/documentation/en/US/Red_Hat_ Enterprise_Linux/6/pdf/Resource_Management_Guide/Red_Hat_Enterprise_Linux-6-Resource_Management_Guide-en-US.pdf.
10. Michael Kerrisk, “Namespaces in Operation, Part 1: Namespaces Overview”, http://lwn.net/articles /531114/.
11. ARM, “ARM Virtualization Extensions”, http://www.arm.com/products/processors /technologies/virtualization-extensions.php/.

12. ARMvisor Project, https://sites.google.com/a/sslab.cs.nthu.edu.tw/armvisor.
13. XEN ARM PV Project, http://wiki.xenproject.org/wiki/XenARM.
14. KVM ARM Project, http://systems.cs.columbia.edu/projects/kvm-arm/.
15. XEN ARM PVH, http://www.xenproject.org/developers/teams/arm-hypervisor.html.
16. GloabalPlatform, “TEE System Architecture”, http://www.globalplatform.org/specificationsdevice.asp.
17. Trustonic TEE Service, http://www.trustonic.com/products-services/trusted-execution-environment/.
18. ARM, “Securing the System with TrustZone Ready Program”, http://www.arm.com/files/pdf/Tech_seminar_TrustZone_v7_PUBLIC.pdf.
19. Trusted Computing Group, “TPM MOBILE with Trusted Execution Environment for Comprehensive Mobile Device Security”, http://www.trustedcomputinggroup.org /solutions/mobile_security.
20. KVM Project, http://www.linux-kvm.org/page/Main_Page.
21. QEMU Project, http://wiki.qemu.org/Main_Page.
22. IBM TrouSerS, http://trousers.sourceforge.net/.
23. Linux Kernel Documentation on “cgroups”, https://www.kernel.org/doc/Documentation /cgroups/.
24. BusyBox Project, http://www.busybox.net/.
25. ARM, “ARM Security Technology: Building a Secure System Using TrustZone Technology”, PRD29-GENC-009492C.
26. Open Virtualization Project, http://www.openvirtualization.org/.
27. TOPPERS SafeG Project, http://www.toppers.jp/en/safeg.html.

Acknowledgements

This research was supported by the MSIP (Ministry of Science, ICT&Future Planning), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency) (NIPA-2013-H0301-13-3002).