CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Survey and example of trusted platform (TPM)
Student:	Andrea Holoubková
Supervisor:	Ing. Jiří Buček
Study Programme:	Informatics
Study Branch:	Information Technology
Department:	Department of Computer Systems
Validity:	Until the end of winter semester 2018/19

Instructions

Perform a survey of existing standards for trusted computing with security chips and integrated components, focus mainly on Trusted Platform Modules (TPM) and Trusted Execution Environment (TEE). Create an example usage of TPM in a simple case such as disk encryption. Demonstrate the effect of changes in the platform on the access to the secured resource (disk).

References

Will be provided by the supervisor.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague September 11, 2017

Czech Technical University in Prague

Faculty of Information Technology

Department of Computer Systems

Bachelor's thesis

Survey and example of trusted platform (TPM)

Andrea Holoubková

Supervisor: Ing. Jiří Buček

3rd January 2018

Acknowledgements

I would like to express my sincere thanks to my supervisor Ing. Jiří Buček for his advice and patience with introducing me to disk encryption issues. Thanks to my friend Marian who read my thesis first and helped me with proofreading. Finally, thanks must go to my family and friends for love, inspiration and support throughout my study.

Declaration

In Prague on 3rd January 2018 . . . .

Citation of this thesis

Holoubková, Andrea. *Survey and example of trusted platform (TPM)*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Bakalářská práce se zaměřuje na průzkum řešení pro realizaci důvěryhodné platformy. Podrobnější popis je věnován standardům organizací Trusted Computing Group (TCG) a GlobalPlatform. Jedná se o bezpečnostní čip Trusted Platform Module (TPM) a integrovanou bezpečnou zónou procesoru nazývanou Trusted Execution Environment (TEE). Součástí řešení bakalářské práce je také praktická ukázka použití čipu TPM na běžném počítači pod operačním systémem Linux. Praktická část se věnuje šifrování externího média (USB disku) a uložení klíče do TPM.

Klíčová slova

Trusted Computing, Trusted Platform, Trusted Platform Module, Trusted Execution Environment, šifrování disku, bezpečnost

Abstract

The bachelor thesis focuses on survey of solutions to implement a trusted platform. A more detailed description is devoted to the standards of Trusted Computing Group (TCG) and GlobalPlatform organizations. It concerns a

Trusted Platform Module (TPM) security chip and an integrated secure processor zone called Trusted Execution Environment (TEE). Part of the bachelor thesis is also a practical demonstration of using a TPM chip on a regular PC computer under the Linux operating system. Practical part is devoted to encryption of external media (USB disk) and storing the key in the TPM.

Keywords

Trusted Computing, Trusted Platform, Trusted Platform Module, Trusted Execution Environment, disk encryption, security

Contents

List of Figures

List of Tables

Introduction

Current computer systems are open platforms where users can run many useful programs but there may be a number of malicious programs. Combined with the many sensitive data we store, the need to secure a computer system is required. The main causes of low security are for example the high complexity of today's operating systems, low security hardware support and weak user safety habits.

These problems can be addressed by Trusted Computing (TC). The first technology I am going to deal with is the Trusted Module Platform (TPM) security chip, which aims to protect sensitive data (passwords, keys, certificates). There are more than 100 million TPM chips in the top brands of HP, Dell, Sony, Lenovo, Toshiba and others [12]. The second technology is the Trusted Execution Environment (TEE). It is an environment that allows for secure execution of applications. Hardware-based TEEs have been widely deployed in mobile devices for over a decade [13]. The TEE can be used to ask the user in a secure way to apply a money transfer and is ensured that the confirmation is from a specific device.

The aim of the thesis is to perform a survey of existing standards for trusted computing with security chips and integrated components, focusing mainly on TPM and TEE. As a practical part, to create an example usage of TPM in a case such as disk encryption and demonstrate the effect of changes in the platform on the access to the secured resource (disk). Lenovo ThinkPad x61s with Linux Mint version 18.01 will be available for a practical demonstration. The chip version will be TPM 1.2.

Chapter **1**

Trusted Computing

1.1 Generally about Trusted Computing

According to [14, 15] Trusted Computing means an approach to building computer systems and infrastructure components which:

- Strongly identify themselves - using public key cryptography, involving a secret key strongly tied to the platform itself.
- Strongly identify their current configuration/running software - using cryptographic hashes of object code, and other mechanisms.
- Allow us to make informed decisions about the level of trust to invest in them.

As implies from the name of Trusted Computing (TC), trust is needed to achieve the security. The term trust has for TC according to [16] the following meaning: *"Trust is the expectation that a device will behave in a particular manner for a specific purpose."* The trust itself does not mean safety. In general, we need trusted components to build secure systems. Trustworthy systems are ones that we rely on to have correct and predictable operations [15]. Some goals of Trusted Computing are according to [1] as follows:

- Improve security of existing computer systems
- Reuse existing software modules
- Applicable to different operating systems
- Open architecture
- Efficient portability
- Avoiding potential misuse of trusted computing

The general concept of Trusted Computing refers to a set of technologies that allow the safe run-time on a trusted hardware and software. Support for Trusted Computing requires hardware support. In the context of Trusted Computing, there is a term named trusted platform. It's a set of trusted components in hardware and software that provide trusted security functions. It creates a trusted basis for running applications and provides hardware protection for sensitive data [1]. Comparison of conventional and trusted platform is in the figure 1.1.

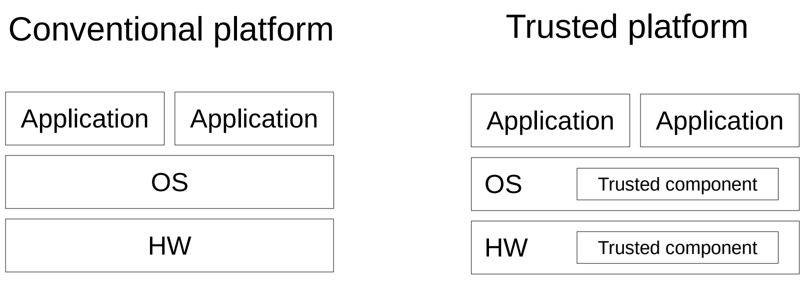


Figure 1.1: Conventional platform vs trusted platform [1]

The following elementary components are required to create a trusted system [1]:

- **Integrity verification (attestation)** - Allows the platform to verify its status.
- **Secure storage** - Securely stores data on untrusted storage.
- **Safe input and output** - Provides safe operations between endpoints.
- **Process isolation** - Provides separate, secure storage space for processes

1.1.1 Approaches to implementation of Trusted Platform

Many types of devices either fit this definition of "trusted computing platform" or have sufficient overlap that we must consider their contribution to the family's lineage [17].

1.1.1.1 Software based attestation schemes

These techniques are based only on software. Some examples are summarized in [18]. The software-based attestation schemes will never give the same confidence level as the hardware mechanisms of a closed platform [18].

1.1.1.2 Secure coprocessors

Probably the purest example of a trusted computing platform [17]. The coprocessor only executes authenticated code and physical shielding provides hardware tamper resistance [18]. Applications running on this computing system can use this isolated environment to achieve a security that cannot otherwise be easily obtained.

1.1.1.2.1 Smart cards

In some sense the latest generation of smart card meets the definition of secure coprocessor. Originally smart cards have been constrained in processing power and storage capacity and a dedicated smart card reader was needed. Application of this cards was limited to some specific tasks (e.g. SIM cards, eID cards, credit cards). Latest generation of smart cards contains high speed interface, higher amout of memory and more powerful microprocessor [18].

1.1.1.2.2 Trusted execution environment

Trusted execution environment (TEE) is a strongly isolated environment inside the main microprocessor. Standards are defined by organization GlobalPlatform. It is used mainly in mobile world [15]. TEE will be described in more detail in section 1.3.

1.1.1.3 Cryptographic accelerators

Special-purpose hardware to off-load cryptographic operations from the main computing engines. It was designed because the deployers of the cryptographic computations (such as banking systems) felt that this machines were not suitable enough for a cryptogtaphy, because of the slow computations of RSA, DSA, Diffie-Hellman when the modulus increases several times. This accelerators has a range from single-chip coprocessors to a bigger stand-alone modules.

1.1.1.4 Dongles

Small device, attached to a machine, that a software vendor provides to ensure the user abides by licensing agreement. It is used to prevent copying the software.

1.1.1.5 Personal tokens

Hardware which user carries to enable authentication, cryptographic operations, or other services. Similar properties as smart cards.

1.1.1.6 Trusted platform module

Trusted platform module is a hardware security chip defined by organization Trusted Computing Group (TCG). It provides basic cryptographic functions for key generation, signing and hash generation. The TPM contains a trusted memory that can only be accessed by itself - it can not be robbed by software. From a security point of view, it acts as the Root of Trust, it must be trusted for all components. The rest of the system is connected to the bus. It can be part of PCs, mobile phones or embedded devices. TPM will be deeply described in section 1.2.

1.1.1.7 Virtualized platforms

In this way a security critical application can run on a dedicated Virtual Machine (VM) isolated from the VM that hosts the legacy operating system. The integrity of the application VM and the hypervisor can be verified with a remote attestation protocol [18]. Running virtual machines can be migrated and copied, and allow for both good utilization of resources and also strong isolation among the contents of the virtualized containers [15].

1.1.2 Examples of use

The use of Trusted Computing is very wide, here are some practical examples of use:

- Network Connection Security - The server will only communicate with a computer that knows based on parameters that can not be modified.
- Platform Integrity - If none of the hardware and software components have been modified by malicious code, it can be considered trusted for running applications.
- Digital Right Management - Restriction of digital content, e.g. can be played only on a specific device.
- Hard disk encryption - the data on the disk is encrypted with a symmetric cipher, the key is securely stored in the trusted hardware.

There is also a criticism of Trusted Computing usage. Some opinions are summarized in [15].

1.2 Trusted Platform Module

TPM is a component described in standards released by organization Trusted Computing Group.

1.2.1 Trusted Computing Group's specification

Trusted Computing Group TPM Main specification, which describes TPM has several versions - 1.1b, 1.2 and 2.0 [19, 20].

1.2.1.1 TCG Main Specification Version 1.1b

TCG Main Specification Version 1.1b was in the form of a release for a few years, and was finally released according to [10, p. 1-2] in 2003 as a TCPA (predecessor of TCG) Main Specification, Version 1.1b. The basic functions (key generation, storage, secure authorization) of a TPM were available. Functionality to help guarantee privacy was available through the anonymous identity keys based on certificates that could be provided with the TPM, for example, owner authorization. The TCG has different specifications with respect to version 1.1b and each addresses the different aspects of the secure design of the system. The disadvantage of TPM 1.1b was incompatibility at the hardware level. TPM vendors had slightly different interfaces requiring different drivers.

1.2.1.2 TCG Main Specification 1.2.

TPM version 1.2 is upgrading and replacing version 1.1b. Therefore, studies of implementations based on TPM version 1.1b can be directly applied to the Specification 1.2 (but not vice versa). Version 1.2 was developed in 2005-2009 and went through several versions. It includes a standard software interface known as a TIS (TPM Interface Specification, see 2.3.1) driver which is designed to support TPM from various manufacturers. In TCG Main Specification Version 1.1b there was no protection against an attacker trying to guess the right password. This attack is known as *Dictionary Attack*. The TPM 1.2 specification required TPM protection against these attacks by specifying the wrong password entry limit. The privacy groups complained about the lack of implementation of the Certification Authorities for Personal Data Protection (explained in section 1.2.5.1). The latest version of the specification 1.2 is named TPM Main Specification Level 2 Version 1.2, Revision 116 [19]. TPM 1.2 was deployed on most x86-based client PCs from 2005 on, began to appear on servers around 2008, and eventually appeared on most servers.

1.2.1.3 TCG Main Specification 2.0.

For TPM 2.0, the specification is named Trusted Platform Module Library Specification, Family "2.0", Level 00, Revision 01.38 and was last edited in September 2016 [20]. Its version from 2015 have been approved ad the International Standard ISO/IEC 11889:2015, Parts 1-4, in 2015 [20]. TCG Main Specification 2.0. was created because of the attack on the SHA-1 digest algorithm. The TPM 1.2 architecture has SHA-1 implemented everywhere

[21]. Therefore TCG began work on a TPM 2.0 specification that would not hard code SHA-1 or any other algorithm but rather would incorporate an algorithm identifier that would allow any encryption algorithm without changing the specification. The main reason for the TPM 2.0 was to create TPM 1.2 plus an algorithm identifier. But in the end it was a lot more than just this improvement. It is because, the TPM 1.1b and 1.2 versions were compact enough to be encrypted with a 2048-bit RSA key in a **single** encryption. This means that there were only 2 048 bits to work with and no symmetric algorithms were implemented. TPM 2.0. had to provide for algorithms that were larger than SHA-1's 20 bytes, making the existing structures very large in comparison to 2048 bits (256 bytes) in previous versions. RSA key could not use encrypt the large structures in single operation, and multiple use of encryption was too slow. Bigger size of the RSA key would mean using sizes of key that were not in supported in the industry and would also change the key structures, and slow down the chip. TPM Work Group wrote specification TPM 2.0 which allowed to use new encryption: Encrypting a symmetric key with the asymmetric key and the data with the symmetric key. *"Symmetric key operations are well suited for encrypting large byte streams, because they are much faster than asymmetric operations. Symmetric key encryption thus removed the barrier on the size of structures. This lead manufacturers to create several functions that were not included in TPM 1.2"* [10].

1.2.2 HW architecture

General HW architecture of TPM is in figure 1.2. It consists of these blocks:

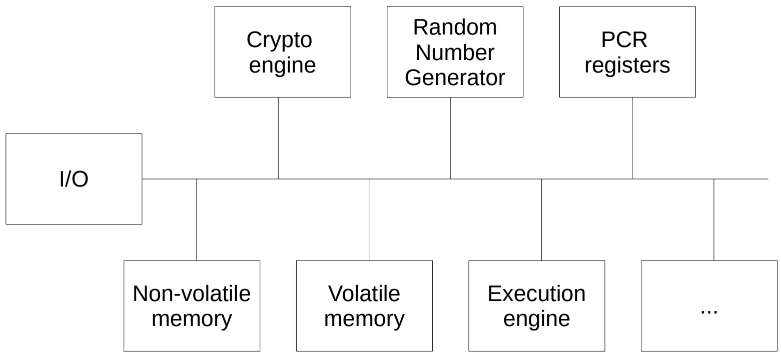


Figure 1.2: TPM architecture [2]

- **I/O component** is a gatekeeper mechanism that manages the information flow in the communication bus. TPM chip is connected to the board via the LPC bus or the SMBus [22]. The TPM chip behaves only as a ”slave”, it only responds to commands that come from the TPM Device Driver.

- **Execution engine** runs program code to execute the TPM commands received from the I/O port.
- **Crypto Engine** performs encryption and decryption algorithms in the chip. According to the specification of TPM 1.2, a RSA , SHA-1 , and a HMAC engines are implemented and also, a keys which are 512, 1024 or 2048 bits long are required.
- **Volatile storage (RAM)** - memory that is dependent on the power supply and the contents inside will be lost during the disconnection time. It holds the current state of the chip, cryptographic keys, authentication session - it is used to store temporary data, i.e. the most frequently cryptographic keys currently loaded.
- **Non-volatile storage (NVRAM, flash)** - memory, whose content remain inside even during the interruption of the power supply.
- **Random Number Generator (RNG)** - serves as the basis for the encryption algorithm.
- **Platform Configuration Registers** - registers for measurements of the platform. These registers are deeply discussed in the separate section 1.2.4.

1.2.3 Keys

The non-volatile storage stores the following data and these main keys:

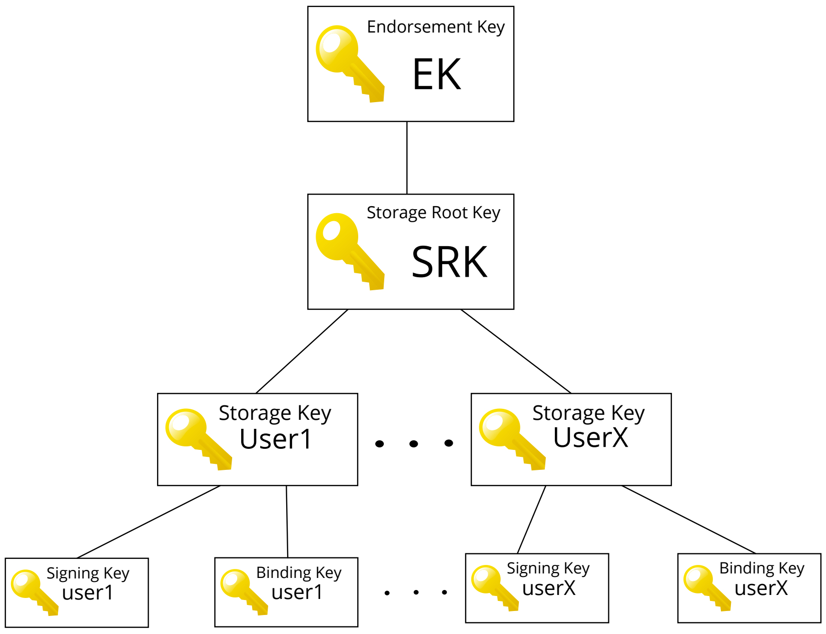


Figure 1.3: Tree hierarchy of TPM keys [3]

- **Endorsement Key (EK)** - A pair of RSA keys that are set by the manufacturer. The private part of the key is never disclosed. It remains inside the chip throughout its lifetime. Its size is 2048 bits.

- **Storage Root Key (SRK)** – It is created when the user or administrator takes ownership over the system by using the `TPM_TakeOwnerShip` command. This key pair is generated by the TPM chip based on the owner password specification, and never leaves the chip. Additional storage keys can be saved. A tree hierarchy is created where the root is the SRK that covers the other keys. It is generated every time the chip changes the owner.

- **Attestation Identity Keys (AIK)** - Protects a device from unauthorized modification of firmware and software by having critical parts of firmware and software hashed before they are executed. When the system tries to connect to a network, hash values are sent to the server to verify that they match the expected values. If they do not match, the system will not be able to access the network. It is stored in non-volatile memory of TPM. It is used to sign the PCR values, which will be discussed in section 1.2.4. Confirms integrity reports. It can replace the EK, but if we sign a message with EK, it would mean that we would always disclose our identity, it would be clear which TPM chip signed the message. Usually, it's enough to prove that the signature comes from a some TPM chip.

The keys form the tree structure where the root is EK. They are divided according to purpose as follow:

- **Storage key** - Used to store additional keys. The child key is always encrypted by the public key of the storage key. For the loading a key to the TPM, its master storage key must be pre-loaded and the authorization to use this master key (password) must be completed. It is a RSA key that serves only to decrypt other keys (Root of Trust for Storage).
- **Binding key** - It is a public RSA key that is used to encrypt a small amount of messages, data, or other symmetric keys to encrypt a large amount of data. It can not be used to encrypt other RSA keys [4]. Used for binding, which will be described in section 1.2.6.
- **Signing key** - To ensure the integrity of any files stored on the platform. The private key part can only be used for signing.

Keys are also divided by the possibility of moving to another TPM. These keys have flags:

- **Non-migrable key** - This key can not be migrated to another TPM chip. Their private parts never leave the key. It is safer than a migrable key. The only risk is the burning of the motherboard. Its data can not be undone, so they will be lost forever. However, with its advantages comes the disadvantages in the form of reduced flexibility (e.g. impossible to decrypt the file on another PC platform with another TPM chip with the same owner). Non-migrable keys include EK and SRK.
- **Migrable key** - This key can be migrated to another TPM chip. There is an increased risk of an operation called *migration*. The owner will provide another public key to the TPM chip, then he may have access to other secret keys. All these operations must be performed by the owner.

1.2.4 Platform Configuration Registers

In volatile memory, the TPM has a minimum of 16 (the newer have 24, e.g., Infineon SLB 9635 TPM) Platform Configuration Registers (PCR). Each of them has a size of 20 bytes because the SHA-1 hash has 20 bytes. These registers store platform integrity measurement results. Here is meant a platform consisting of several components. Typical components are BIOS, Master Boot

Record, Boot Sectors, The Boot Loader, and finally the operating system and application software. The component can measure another component (counts its hash) and puts this measurement into the PCR (before it passes control to that component). PCR p is extended with m by the following formula:

$$p := \text{SHA-1}(p||m)$$

This insertion is an irreversible process and is called *extend*. PCR is used for secure boot (Trusted Boot). We can not directly write value to them, we can only extend the value by a new one. The entire measurement history is stored there. The consequence of this functionality is that neither we nor the attacker can write down the specific value, so that the measurements that are stored in the PCR can not be additionally falsified.

The table 1.1 shows what measurements are stored in the PCR registers.

PCR	Allocation
0	BIOS
1	BIOS configuration
2	Option ROMS
3	Option ROMS Configuration
4	MBR (Master Boot Record)
5	MBR Configuration
6	State transitions and wake events
7	Platform manufacturer specific measurements
8-15	Static operating system
16	Debug
23	Application Support

Table 1.1: Measurements of PCR registers [10]

1.2.4.1 Access to the PCR

PCR needs a 32-bit index, which determines the maximum usable PCR index. All indexes beginning with the 230, have been reserved by TCG for higher or later versions of the specification.

PCRs and the protected capabilities that operate upon them may not be used until Power-On Self-Test (TPM POST) has completed. If TPM POST fails, the `TPM_Extend` operation will fail. In addition, the `TPM_Quote` operation and `TPM_Seal` operations that report and control the PCR values also fail. After successful TPM POST execution, all PCRs have to be set to preset values [23].

1.2.4.2 Usage of individual configuration registers

PCRs from 0 to 7 are used for SRTM (see section 1.2.5.3). They are defined for use within the Pre-Operating System state. Their resetting is only possible when the whole platform is reset. During the BIOS boot process a log of all executable code is created and extended into PCRs. [24] The other 8 registers (PCR 8 to 15) are used after the operating system has been deployed (Operating System Present State). The CRTM may measure itself to PCR 0 and POST BIOS, including manufacturer-controlled embedded Option ROMs, firmware, etc., that are provided as part of the motherboard. Only executable code is logged. Configuration data such as ESCD (Extended System Configuration Data) should not be measured as part of this PCR [23, 24].

The motherboard and other hardware configuration metrics are located in PCR 1.

Measurements made over optional ROM (Option ROM) devices that are not part of the motherboard (Non-Host Platform Adapters) are placed in PCR 2. There are two options of Option ROMs: visible and hidden. Each is measured and logged to PCR 2. Visible must be measured by the BIOS. Option ROM is responsible for measuring the hidden code. The code must be measured before it is executed. Option ROM may have configuration and other data that are relevant to the trust properties of the Host Platform. For example an SCSI controller's configuration of its hard disks, e.g., RAID type, drive assignments is measured into PCR 3 [24].

PCR 4 measures the process of attempting to boot different hardware paths , e.g. from a DVD or a hard drive, and the IPL Code that is loaded and executed from the device. Information about which paths was selected will be recorded by the IPL code in the PCR 5. PCR 6 contains measurements of the Platform Manufacturer during boot and later by an OS. Platform Manufacturers use the PCR to measure what turned on the platform. PCR 7 is reserved for platform manufacturer applications. Which means that the platform manufacturer's application can use this PCR before the Pre-Operating System State is introduced.

PCR 16 is for debugging. It is is for use by any entity on the Host Platform.

PCR 17-23 is used for DRTM (see 1.2.5.3.2).

1.2.5 Root of Trust

A set of features that are always trusted by the OS. It is a component that must be trusted implicitly because it is not possible to verify that it is doing what is expected of it. It is divided into:

- **Root of Trust for Measurement**
- **Root of Trust for Storage**
- **Root of Trust for Reporting**

Before individual components are analyzed, it will be explained the term *Attestation*. According to [4], the *Attestation* is the means by which a trusted computer assures a remote computer of its trustworthy status. In connection with the TPM chip, the term *Remote Attestation* is used. The TPM is manufactured with an EK key (see section 1.2.3) built into TPM hardware that contains the public and private section. The public part of the EK is certified by the appropriate certification authority as the EK of a particular TPM chip. Each individual TPM has a unique EK, so it can sign a statement of its trustworthy status using its private part. The remote computer can verify that these statements have been signed by a trusted TPM. However, the anonymity of the TPM is lost, because the EK is its unique identifier and if a digital signature was performed by the EK, then it can be tracked. So the use of the EK as a signature does not ensure privacy.
The attestation identification key (AIK see section 1.2.3) is the solution. AIK is the key pair created during the attestation for using of a concrete application. It is used instead of EK for several reasons:

1. Reduces direct access to TPM
2. Helps to prevent EK cryptonalysis
3. Solves privacy issues because AIK is not directly connected to hardware

1.2.5.1 Root of Trust for Reporting

Root of Trust for Reporting (RTR) is in charge of a trusted status report. As noted above, a remote attestation or attestation, is a method that proves to a remote person that a local computer is a trusted platform and displays its current configuration of registers. The key to this method is RTR. To avoid this key being EK (due to the anonymity of the TPM chip), the AIK key is generated to sign the configuration. The AIK key has the same properties for the verifier as the EK.

According to the version 1.2 specification, this cryptographic primitive is only present from this version and I will continue to build on it.

To allow an anonymous attestation, TCG has adopted two different approaches to enable it:

- **Use of a certification authority**
  AIK, which is signed by the EK, is first sent to a trusted partner, called Certification Authority (CA). He checks the signature and status of the EK and signs AIK. The remote computer will see one AIK signed by the CA and can not connect it to the EK. New AIKs will be generated for new needs so they can not be interconnected. For a better understanding, here is a picture illustrating this method:

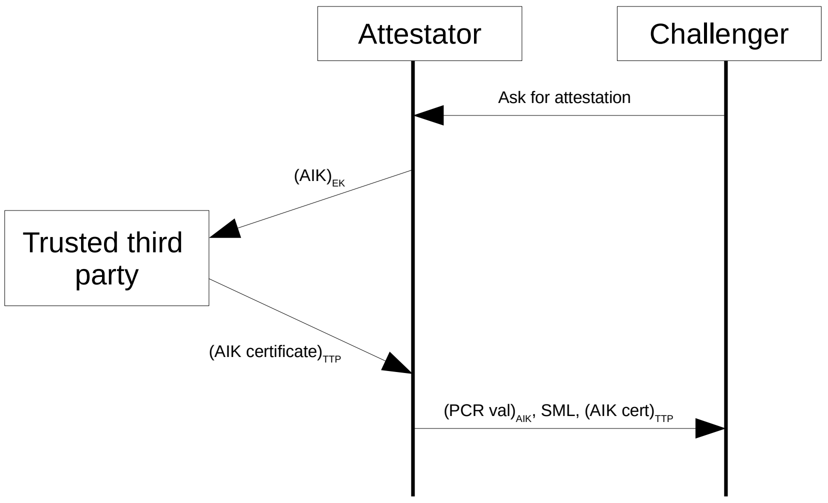

Attestator
Challenger
Ask for attestation
$(AIK)_{EK}$
Trusted third party
$(AIK\ certificate)_{TTP}$
$(PCR\ val)_{AIK}$, SML, $(AIK\ cert)_{TTP}$

Figure 1.4: Attestation with the help of a Certification Authority [4]

  The Attestator generates a key pair of AIK public and private key upon receipt of a Challenger request and sends the public part signed by its EK to a trusted third party (TTP in the picture 1.4) called Certification Authority. It verifies its EK and generates an AIK certificate. The certificate is signed by the TTP and sent back to the confirmation. The Attestator can now send the Challenger its PCR values (signed by AIK), its Stored Measurement Log (SML), and the received AIK certificate.

- **Direct Anonymous Attestation**
  Complicated protocol that does not use CA but is also anonymous. For more information, read here [25].

1.2.5.2 Root of Trust for Storage

Root of Trust for Storage (RTS) is responsible for protecting the external TPM objects that lie in the unprotected memory (e.g. AIK is stored on the disk but is encrypted with a key stored in the TPM) due to a small memory of the TPM. Each key is encrypted with its parent key (see figure 1.3). The root of this tree hierarchy is the EK, which is permanently stored in the TPM (non-migrable key). Other keys are generated by the TPM, encrypted with some keys, and then stored on the disk. In order to use a key, for example, to decrypt, we need to use its parent keys together with it. This entire decryption process ends with the EK, using a key stored directly in the TPM, which is considered to be Root of Trust for Storage.

1.2.5.3 Root of Trust for Measurement

Root of Trust for Measurement, (RTM) is an initialization process that is capable of performing a reliable integrity measurement. It consists of Core Root of Trust for Measurements (CRTM) and component. The credibility of RTM is based on platform certification.
CRTM is part of the unmodifiable and indelible part of the BIOS. The CRTM is the initialization part of the code that is executed on the platform when the computer starts up. It runs before it loads the BIOS so it can measure all its components before it passes control. It measures the rest of the BIOS, calculates its cryptographic hash, writes the result to the PCR register, then boots the BIOS and initializes it. The BIOS measures the hardware configuration as shown in Fig. 1.5:

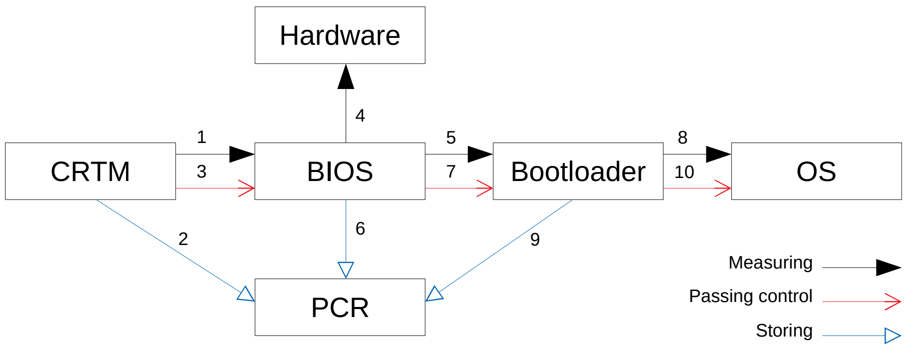


Figure 1.5: Measurement process [4]

Next it measures the bootloader of the operating system. It retrieves it from the disk, calculates its hash, and store a calculation to the PCR registers using the *extend* operation. Then the bootloader starts, which measures the kernel OS and also saves the result in the PCR register before it starts. We can see all the measured components and their cryptographic sums by checking the status of the PCR registers and see if the OS is running in the expected state or not.

There are two ways of how the Root of Trust for Measurements can be measured. These methods are named Static Root of Trust for Measurements (SRTM) and Dynamic Root of Trust for Measurements (DRTM).

1.2.5.3.1 Static Root of Trust for Measurements (SRTM)

SRTM occures at a system boot. The first thing being processed at boot is the static, unchangeable piece of code called Core Root of Trust for Measurements (CRTM). Trusted Platform needs an additional entity, which would measure the BIOS itself and act as a CRTM. This entity is called a Trusted Building Block (TBB) which remains unchanged during the lifetime of the platform.
There are two ways to integrate the CRTM with the BIOS. First, the whole CRTM code is contained in the BIOS boot block, so the rest of the BIOS can be updated. The inconvenience of this way is that rest of the BIOS has to be measured by CRTM for integrity. Second way is that whole BIOS becomes a part of TBB, so the whole BIOS is CRTM. Any change to any of this requires new measurements

1.2.5.3.2 Dynamic Root of Trust for Measurements (DRTM)

The term was introduced in specification 1.2. This method addresses the disadvantage of the SRTM, which is that many modern systems (e.g. Microsoft Windows and Linux) require executable content to be measured (executable libraries, shell scripts etc.). Those components which were once measured are often updating while the system is running, and also some of the elements may be executed in different orders, which gives rise to different measurement values in PCRs. DRTM allows launch of the measured environment at any time without a platform reset. So the goal of DRTM is to create a trusted environment from an untrusted state. Implementation of Intel is called Trusted Execution Technology (TXT) while AMD use the name Secure Virtual Machine (SVM). DRTM uses PCRs 17-22. Command `SENTER` is one of a new instruction introduced by TXT (SMX operations - CPU instructions). It is used for TXT late launch from untrusted to a trusted system without reboot. These instructions enable some pieces of code to be executed in an isolated environment and cryptographic hash of a code is calculated before execution. The hash is extended into a PCR on TPM chip. And it is always loaded dynamically.
More information about this technique can be found on [8].

1.2.6 Classes of protected message exchange

Keys are communication endpoints and improperly managed keys can result in loss of security. TCG defines the four classes of protected message exchange [26]:

1.2.6.1 Binding

Binding is the encryption of a public key message, called *binding key*, which is stored in the TPM chip. If the binding key is non-migrable, the corresponding private key can only use the TPM where the key was created. Therefore, the encrypted object is effectively bound to a specific TPM.

1.2.6.2 Sealing

Sealing is an extension of the binding operation. Not only is the encrypted message tied to a particular TPM, it can also be decrypted only if the current configuration of the platform matches the PCR values at the time of creation. Once the data has been decrypted, the TPM verifies that the system status is the same as the values recorded by the PCR. If so, it gives the data to the user.
This is one of the most powerful ways to use the TPM because protected messages can only be viewed if we have a functional platform and know its specific features in advance.

1.2.6.3 Signing

Signing is an operation that ensures data integrity.

1.2.6.4 Sealed Signing

Sealed Signing are operations that extends signing. These operations are linked to the PCR registers to make sure that the platform that signed the report is in a state that matches its requirements.

1.2.7 TCG Software Stack

The Trusted Software Stack (TSS) has been defined by the TCG to make the TPM chip functions available to the program. It provides a standard API and extends limited chip capabilities. TSS simplifies working with TPM and provides additional services:

- Ability to store keys on disk
- Remote TPM access through the network
- Data conversion between portable formats (data blocks)

The stack according to the standard has many layers and interfaces. On the lowest layer is the TPM chip driver and on the top layer is the service library that is called from the application programs. In this section we will describe the TSS architecture from the lowest level to the highest. The TSS architecture is shown in Fig. 1.6.

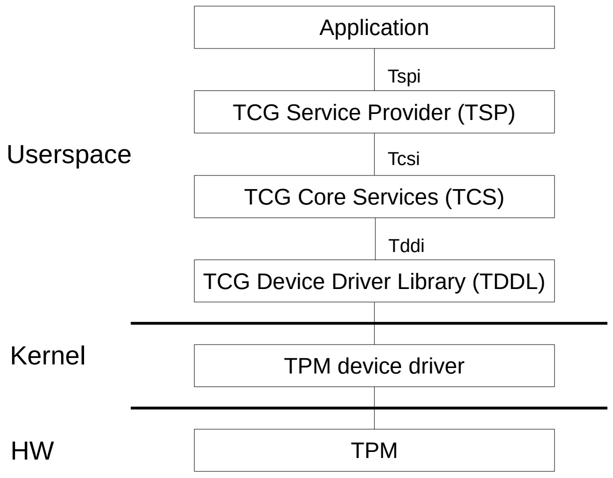


Figure 1.6: TSS layers [2]

1.2.7.1 Description of the TSS architecture

On the lowest layer of hardware is the TPM chip that communicates with the TPM Device Driver in the kernel. The TCG Device Driver Library (TDDL) is on the lowest layer of userspace. This library communicates with the device driver inside the kernel.

1.2.7.1.1 TPM Device Driver

The TPM Device Driver (TDD) is located in the kernel of the operating system. It is a software that communicates with the TPM via the LPC bus where the TPM chip is connected to the rest of the system. The driver receives messages from the top of the TCG Device Driver Library (TDDL) and sends it back to the TPM. TDDL contains only 4 commands - `open`, `close`, `transmit`, `receive`. It is used to connect the TCS daemon to it. In the TCG specification is assumed to run in a kernel mode, because in most operating systems, the user application has no direct access to hardware that is necessary for TDD. TDDL does not deal with Multithreading.

1.2.7.1.2 TCG Core Services

TCG Core Services (TCS) runs as a daemon, which is the only process in the system that accesses the TPM through the TDDL and its Tddli interface. TCS resolves efficient TPM resource management, manages resources, and switches TPM contexts. Because the TPM can not handle commands in parallel, the solution to this phenomenon has been implemented at the TCS level. The TPM requires TCS to be run as an operating system service if it is in the platform's capabilities. TDDL provides only one TDD connection, so there can be no more than one TCS service. The TCS interface is Tcsi.

1.2.7.1.3 TCG Service Provider

TCG Service Provider (TSP) is a shared library that links applications. It is a module that provides an object-oriented interface for applications where we want to use all the TPM options. It accesses TCS and has some of its own services. Provides an individual key store for users. The interface used by TSP access applications is Tspi. Tspi is an interface in C language. TSP provides additional TPM auxiliary functions, such as signature verification.

1.3 Trusted Execution Environment

1.3.1 Specification

The Trusted Execution Environment (TEE) has about 12 published specifications, which are maintained by GlobalPlatform. GlobalPlatform is a non-profit organization that develops security-related specifications. It has more than 120 members (eg AMD, Apple, ARM). Technologies from this organization are globally used in many areas such as finance, telecommunications, cars, etc. All specifications are published on GlobalPlatform website [27].

TEE is a technology that allows running security applications independently of the operating system (isolated run), after implementation in the processor. A protected application is not accessible by any software tool from a major operating system, which is more vulnerable in terms of safety. Every specification contain an important information about this technology.

- **TEE Client API** [27, TEE Client API Specification v1.0] specification is a low level communication interface which enables access of Client application running in the Rich OS to exchange data with a Trusted Application running inside a Trusted Execution Environment. The

- **TEE System Architecture** [27, TEE System Architecture v1.1] defines the hardware and software architecture that constitutes the TEE.

- **TEE Internal Core API** [27, TEE Internal Core API Specification v.1.1.1] defines interfaces available within a TEE, such as memory management, cryptographic primitives and secure storage for the TA development. The definition of others specification is part of the GlobalPlatform website, which is also an initializer of standardization. Standardizing is very important, because every company which creates TEE has another architectural option.

1.3.2 Fundamentals of TEE

TEE serves to secure a trusted platform on the smartphone and uses isolation principle of the the critical application from the rest of the system. OSs for smartphones are designed to be more secure than OSs for PCs but security is routinely defeated using techniques such as *jailbreaking* on iOS and *rooting* [13, p. 159–178] on Android OS. On a common operating system in the TEE terminology called the Rich Execution Environment (REE), a number of applications may run. Possible malicious applications may disrupt the security of other applications in this environment. To implement TEE technology, support is required directly in the processor. Specific implementations of TEE vary among processors manufacturers. The Intel processor manufacturer calls it Intel Software Guard Extensions (SGX), the manufacturer ARM calls it TrustZone. TEE technology is mainly used in mobile phones and embedded devices. Because the dominant processors on these platforms are from the manufacturer ARM, this thesis will focus on the TrustZone technology.

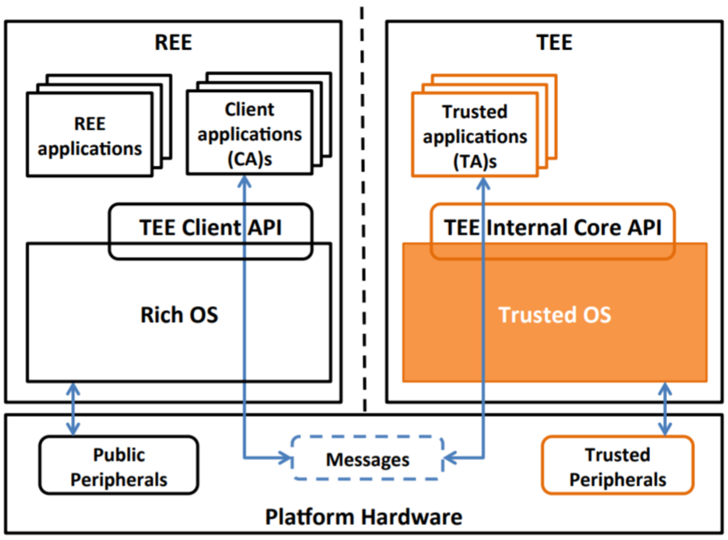


Figure 1.7: GlobalPlatform's TEE system Architecture [5]

Using the terminology introduced by GlobalPlatform [18], Figure 1.7 describes the concepts of TEE.

1.3.2.1 Rich Execution Environment

The word ”rich” means an operating environment which is hugely extensible but they are more prone to being attacked for example modern platforms such as Android, iOS, Microsoft Windows or Linux, where runs the majority of the platform software, including the primary operating system [27, TEE System Architecture v1.1].

1.3.2.2 Trusted application

An application logic that runs inside the TEE is referred to as a *trusted application* (TA). It provides security related functionality to Client Application (for example a service style application which provides a cryptographic keystore) [27, TEE System Architecture v1.1].

1.3.2.3 Client application

The REE application, which initiates and interacts with the the TA, is called a client application (CA)[1]. It is for example browser or e-mail client.

1.3.3 Use of TEE

TEE can efficiently protect sensitive data, however, it is not impermeable. User can downloaded a new update for their device which contains malware, which can masquerade as a legitimate CA, therefore the attacker can freely use the sensitive data stored in the TEE and decrypt data as a super user, however, the TEE would still hold the key which will not be revealed. Despite these shortcomings, TEE and its functions still prevail the risks of not using it [5]. TEE is widely used for mobile devices, which contains these functions: [13], [28]:

- **Secure Boot** CPU boot in ”secure kernel mode” in ROM. The ROM itself is a ROM Boot Loader and it is booting in a mode called Secure Boot. ROM Boot Loader has a digital signature of TEE OS. Before it passes execution to the TEE OS, it will verify the signature. Then it starts executing the TEE and the TEE itself will verify Rich OS integrity with digital signature. Basically, each stage of the boot sequence verifies the signature associated with the next stage software before loading it into the secure world. As long as each stage verifies the signature of the code image of the next stage, only authenticated code can be executed. One of these loads a non-secure, non-authenticated bootloader in the normal world which in turn loads the main operating system [6]. Secure boot does not guarantee that the device is free of security issues; rather it can certify that the components that have booted are the best known configuration as provided by a trusted source. The idea of the TEE separation is to apply secure boot to the TEE, but not necessarily to the main OS.

[1]CA is widely used as a shortcut for Certificate Authority.

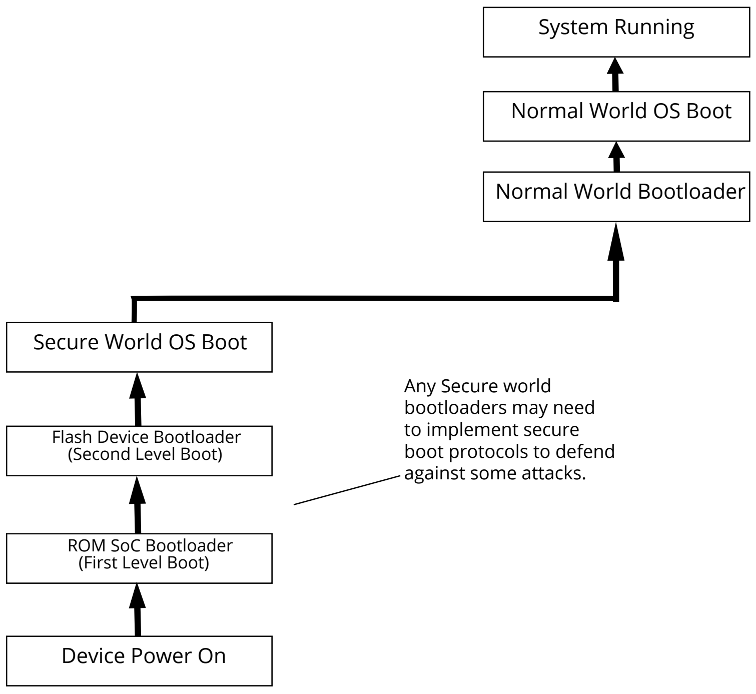

System Running
Normal World OS Boot
Normal World Bootloader
Secure World OS Boot
Flash Device Bootloader (Second Level Boot)
ROM SoC Bootloader (First Level Boot)
Device Power On
Any Secure world bootloaders may need to implement secure boot protocols to defend against some attacks.

Figure 1.8: Typical TrustZone boot process [6]

- **Secure storage** Secure storage is used to protect the integrity of the data while they are not in use.

- **Remote attestation** For remote verifying the integrity of the service that is actually talking to the software on the device. It allows end points that communicate with a secure execution environment to verify the authenticity of the software and hardware implemented by the TEE.

- **Trusted path** Sending and receiving communication from and to the outside world while guaranteeing the authenticity of the communicated data and optionally also the secrecy and availability should be possible. This should allow the one of the end point, whether human or not, to verify that the data transmitted comes from the execution environment and also it should ensure that data from peripherals received in the environment is authentic [28].

- **Digital Rights Managements (DRM):** Media industry uses TEE widely. For protecting the media stream from piracy the data is encrypted with a key that is generally device or session specific. The TEE protects this key and can safely display the data in the other part of the system. Taken from the GlobalPlatform website: *"The TEE is the ideal environment for content providers offering a video for a limited period* *of time that need to keep their premium content (e.g. HD video) secure so that it cannot be shared for free [29].”*

- **Key storage:** TEE can be used for a storage of cryptographic keys or certificates, which makes it more difficult to extract them from the device. When keys are stored in a TEE, a CA can make use of them through a key-store Application Programming Interface (API) and these tokens are not exposed to user space [5].

One example of using TEE is in mobile banking. Since many sensitive use cases lead to an interaction with the user, they are mainly related to bill payment, money transfer etc. Mobile banking application communicates with a back-end of a bank. These use cases forms a whole application following these requirements:

- No one should have access to the execution of the banking application and the data stored in it. To achieve this, it is necessary to have a secure storage.
- Instance of the application and the back-end should be able to securely communicate with each other. Remote authentication provides the ability to authenticate the application. Secure storage is used to store credentials needed to setup a secure and authenticated connection with the back-end.
- The user should confirm the transaction safely on its display without anyone interfering. Display should with the trusted path show the transaction details and allow user to securely confirm a transaction by entering a PIN.

1.3.4 Architecture of the TEE

There are different ways to achieve a TEE.

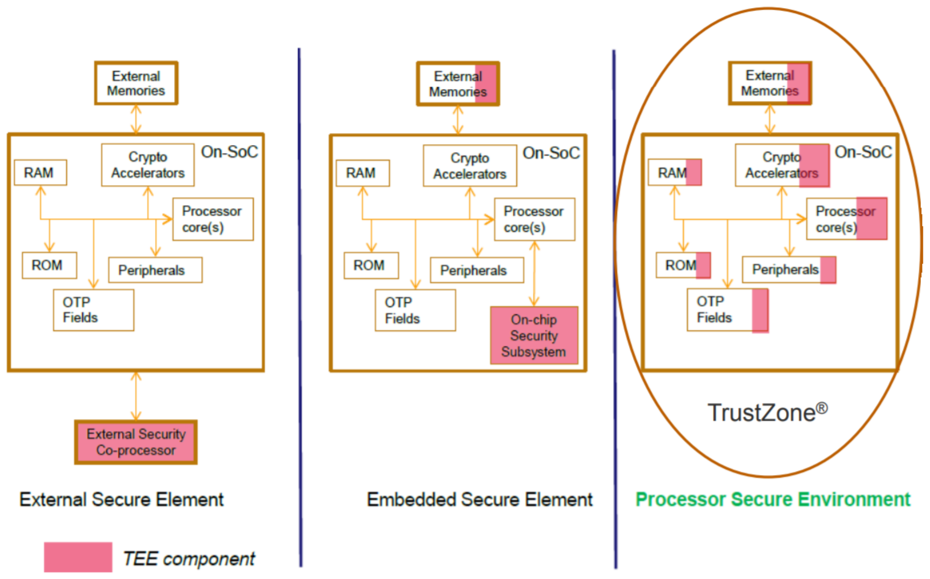


Figure 1.9: Three potential architectural options for realizing a TEE architecture (adapted from [6])

1.3.4.1 TEE component as a coprocessor

A coprocessor is a computer processor used to supplement the functions of the primary processor. It is a separate core with its own peripherals, is used to remove tasks which are critical for security from the main processor. Its advantage is that its operation can be completely isolated from the rest of the environment, and running with the main core, but the drawbacks are transferring data to and from the core which can slow down the entire operation. It is less powerful than the main core. It can be divided as follows [5]:

- **External security coprocessor** is a coprocessor outside the physical chip (commonly referred to as "System on Chip" or SoC) containing the main core, where the coprocessor is isolated from it, and do not share any resources with the SoC and provides no functionality.
- **Embedded security coprocessor** is embedded into the main SoC and has the capability to share some of the resources of the main system. It is still isolated from the main processor. These two processors will not have the same performance (coprocessor will be slower) and it will not be able to interact in a same way.

Because none of the options is sufficient as a TEE component, the TEE has to be in each part of the whole processor and share all the resources. This configuration is referred to as the **Processor Secure environment**.

1.3.5 ARM TrustZone

ARM TrustZone is one of these configurations. It is a technology that adds security features to the ARM cores. An overview of ARM TrustZone is given in a white paper [6] by ARM. In this section an overview of ARM TrustZone based on this white paper will be given.

TrustZone allows us have on the same CPU two parallel worlds - ”Normal world” and ”Secure world” logically separated to be running at the same time. So if we run, for example, an Android application and exists a ”Secure world”, the only way for us to determine that there is another one running is to do parallel measurements. Both domains have the same capabilities, but operate in a separate memory space. It enables a single physical processor core to execute from both the Normal world and the Secure world, where Normal world components cannot access secure world resources. TrustZone is implemented in Cortex-A5/A7/A8/A9/A15, and ARMv8 64-bit Cortex-A53 and A57. Smartphones and netbooks typically utilize this line of processors. Below are a few usage examples for TrustZone as listed by ARM [30]:

- Secured PIN entry for enhanced user authentication in mobile payments and banking
- Software license management
- Access control of cloud-based documents

One advantage is that no additional security hardware, such as cryptoprocessor, is needed.

1.3.5.1 Switching the worlds

TrustZone relies on the so-called NS (Non-Secure) bit. The value of NS bit is transferred throughout the AMBA3 AXI system bus to separate the execution between a secure world and a non-secure world [6]. The NS bit informs a system which mode the processor is currently running in. If the NS-bit is set to 1, the processor is working in a non secure mode and is not able to access secure memory or devices. If the NS-bit is set to 0, the processor is working in a secure mode and is able to access any hardware within the system. The NS bit value in a special register called the *Secure Configuration Register* indicates whether the processor executes in the secure or normal world at any given moment.

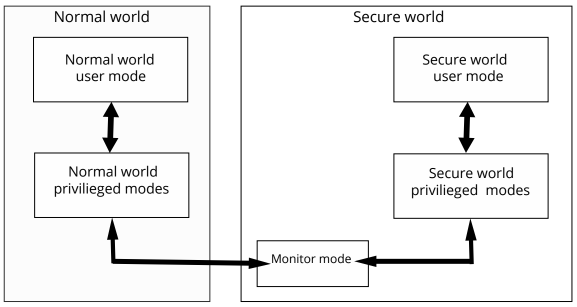


Figure 1.10: Modes in an ARM core implementing the Security Extensions [6]

Switching from one world to another can be compared to a regular context switch. To switch the context between worlds, we need a mechanism known as monitor mode [6]. It is a small piece of software which will make sure neither processor will have access to registers belonging to the other after a switch between the worlds. The monitor also prevents the untrusted processor from using secure memory or devices by managing NS bit. Secure monitor is only running in monitor mode and only a trusted processor can run in monitor mode. An instruction *Secure Monitor Call* (SMC) is used to trigger the processor running in normal world to enter ”monitor mode” that gives the enter to the secure world.

Monitor stores all registers of the currently running world and loads previously stored registers of the world into which is switching to.

Because the monitor is trusted and runs in the secure world, it can store the secure world’s register information in its own address space without risk of leakage of information. The secure monitor call must be used in order to enter the monitor and switch to the secure world, because CA need to access functions of TA. A supervisor call, SVC, must first be made, because the secure monitor call can only be used when in **supervisor mode**[2]. When returning to normal world, everything must be repeated the same way. If the secure world does not have a user mode, a supervisor call may not be necessary.

1.3.6 Other techniques

There are a number of techniques that also can provide a TEE, not only just a secure hardware. I will cover one of them in this section and also I will refer to [31, p. 32, section 2.5] for the others.

[2]A privileged mode entered whenever the CPU is reset or when a SVC (Supervisor Call) instruction is executed.

1.3.6.1 Software

Using TEE only in software is also possible, even though this solution does not coincide with the definition that there is no hardware separation. I am including it here just for completeness. There is an open source solution called Open-TEE, a virtual TEE implemented in software, which follows GP specifications. As it is written on its page: *"Primary motivation for the virtual TEE is to use it as a tool for developers of Trusted Applications and researchers interested in using TEEs or building new protocols and systems on top of it [32]."*

1.3.7 TEE enabled devices and a need for standardization

Commercial implementation of TEE that has been qualified by GlobalPlatform, can be found here [33] and many other TEE products offer a high level of compatibility with GlobalPlatform standards, such as Sony, Toshiba, HTC, Lenovo, etc. Even platforms using the same type of TEE are often not able to exchange and make use of information. An application written for one TrustZone-based platform does not have to run on a different TrustZone-based platform. They may be using different TEE OSs or different REE OS drivers. Developers are less interested in the complexity of low-level software or hardware, but they are more occupied with the problem of using TEE capabilities easily and through different vendors and platforms which would need a standardization.

Detailed information about standardization can be found in [5, p. 13, section 2.4].

1.4 Conclusion

Both of these trusted platform specifications are a way how to secure computing devices. Both are designed on a hardware-based method. While TPM is a piece of hardware specifically created to do cryptographic calculation and is physically isolated from the rest of the processing system as a separate integrated circuit on the mainboard, the TEE is an secure area inside a SoC. Both share the common goal and have a lot of the same elements - secure boot, secure storage, remote attestation etc. The TEE can also be used on PC platforms, but more practical use is in mobile devices (money transfer, mobile payment). TPM is mainly for a PC platform, but can be used also for a mobile devices as written in white paper [34] which combines these two technologies.

TPM was described more deeply, because this technology will be used in practical part of this thesis.

Chapter 2

Practical part - Analysis

As a practical demonstration of using the TPM, I have chosen to encrypt USB mass storage device and save a key in the TPM.

Principle of my design is in the figure 2.1. USB mass storage device can be used as a storage for sensitive data. It will not be used to transfer data between other PCs but will be connected to a single trusted computer. The disk data is encrypted with a symmetric cipher. The key to the cipher will be safely stored in the TPM module. It releases the key only if the platform integrity is not compromised, using the PCR registers functions that store the computer configuration. In the case of disk lost, theft or trusted PC attack, the target computer will be untrusted. There is no key for decryption and the data on the encrypted disk are protected against misuse.

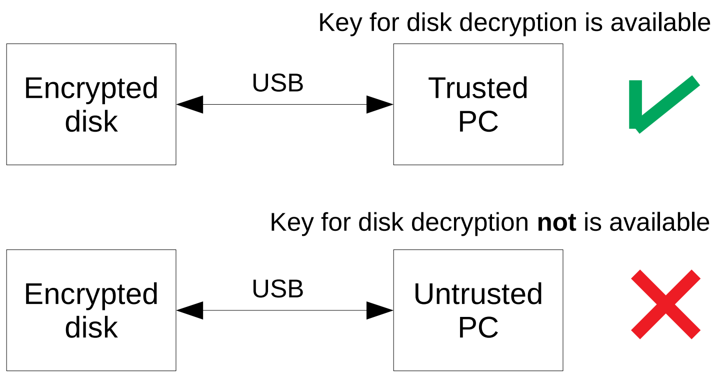

Key for disk decryption is available
Encrypted disk
USB
Trusted PC
Key for disk decryption **not** is available
Encrypted disk
USB
Untrusted PC

Figure 2.1: Principle of practical example

This chapter is a survey and description of tools which will be used for practical demonstration. It deals with disk encryption, trusted bootloader and tools for communication with the TPM.

2.1 Disk encryption tools in Linux

2.1.1 Available methods and tools

In the table 2.1 is a comparison of cryptographic tools used for disk encryption in Linux.

Tool	Loop-AES	dm-crypt +/- LUKS	Truecrypt	eCryptf	EncFs
Type	block device encryption	block device encryption	block device encryption	stacked filesystem encryption	stacked filesystem encryption
Main selling points	longest-existing one; possibly the fastest; works on legacy systems	de-facto standard for block device encryption on Linux; very flexible	very portable, well-polished, self-contained solution	slightly faster than EncFS; individual encrypted files portable between systems	easiest one to use; supports non-root administration
In Linux	must manually compile custom kernel	kernel modules: already shipped with default kernel; tools: device-mapper, cryptsetup [core]	truecrypt 7.1a-2 (read-only features in later versions)	kernel module: already shipped with default kernel; tools: ecryptfs-utils	encfs

Table 2.1: Comparison table of cryptographic tools in Linux [11]

The available disk encryption methods can be separated into two types by their layer of operation:

- **Stacked file-system encryption** (layer that stacks on top of an existing file-system). It is an encryption where individual files or directories are encrypted by the file system itself. All files are written to an encryption-enabled folder which will be encrypted while on-the-fly before the underlying file-system writes them to disk and decrypts them whenever the file-system reads them from disk.

- **Block device encryption** *"method operate below the file-system layer and make sure that everything written to a certain block device is encrypted. This means that while the block device is offline, its whole content looks like a large blob of random data, with no way of determining what kind of file-system and data it contains. Accessing the data happens, again, by mounting the protected container (in this case the block device) to an arbitrary location [11]."*

The method of accessing the data of protected (encrypted) container is possible by a **device-mapper**. Device-mapper provides creating of virtual layers of block devices. The encrypted data can be unlocked by mapping partitions to a new device name. This informs kernel that device is actually an encrypted device and should be used through **dm-crypt** (device-mapper crypt) using the `/dev/mapper/dmname`.

2.1.1.1 Stacked file-system encryption tools

Tools for encrypting file-system are **eCryptfs** and **EncFS** [11].
eCryptfs is an encryption stacked on an existing filesystem, stores metadata in the header of each file.
EncFS creates a transparent on-the-fly encrypted space. This space looks like a regular disk. It encrypts files which will have encrypted all content and a name. Everything is stored back in their directory. Its advantage is that there is no need to create a container[3] in advance.

2.1.1.2 Block device encryption tools

There are a few tools for block encryption in Linux: **Loop-AES, TrueCrypt**, and **dm-crypt**. Loop-AES requires a custom kernel module, it is more work to install, and also is considered less-user friendly. TrueCrypt is not supported for TPM, see manual for TrueCrypt [35, p. 129]. It does not use an authenticity of the platform as a trustfulness. Also, TrueCrypt has stopped development in May 2014. The follower is Veracrypt. The best choice is certainly dm-crypt with LUKS (Linux Unified Key Setup) extension. Others will not be discussed in this work.

2.1.2 dm-crypt/LUKS

Dm-crypt is a specific Linux kernel subsystem versions 2.6 and later that uses the device mapper infrastructure and Linux CryptoAPI to create a transparent layer for online encryption and decryption of block devices.

[3]While using block device encryption, we have to create a container in a file to which we will then store sensitive data. However, this process has the drawback. It's static, it occupies one large disk space and we have to dimension it to the capacity we want in the future. Manipulating with it is difficult.

Linux distributions support the use of dm-crypt on the root file system. These distributions use initrd to prompt the user to enter a passphrase at the console. Most of distributions have dm-crypt included by default. Dm-crypt device mapper target resides in kernel space and relies on user space front-ends known as a **cryptsetup**. It is the interface and command line tool for dm-crypt to create, access and manage encrypted devices.

The tool is used as follows:

```
cryptsetup <OPTIONS> <action> <action-specific-
    ↪ options> <device> <dmname>
```

where `<OPTIONS>` is for example *–key-file* (uses file as a key material) or *–verify-passphrase* (set passphrase) or *–cipher* (cipher specification string). When device is encrypted, it is protected by a key, which can be either a passphrase of a key-file. Both keys have maximum sizes: passphrases can be up to 512 characters and key-files up to size 8192 kiB. The `<action>` is an operation for mapping. Cryptsetup has a five operations for mapping in plain dm-crypt mode - *create, remove, reload, status, resize.* Options `<device>` is the selected device, which we want do encrypt and `<dmname>` is a name for device-mapper.

Cryptsetup has different encryption modes with dm-crypt: LUKS, plain, loopaes and tcrypt (TrueCrypt). Default mode is a LUKS.

- **LUKS** mode is a dm-crypt superstructure that has two-level encryption, key management, and a function PBKDF2[4] which is used for reinforcement of user password. The LUKS specification is multiplatform and can be used to create encrypted containers that can be opened on other operating systems (such as the FreeOTFE application for Microsoft Windows).
  Each disk/partition that is encrypted with LUKS is equipped with a LUKS header with the cipher, hash, encryption mode, and master key checksum.

  Two-level encryption for LUKS means that the volume itself is encrypted by a randomly generated master key, which is then encrypted by a user password and stored in one of the 8 slots, each of which has a LUKS header. On the one hand, it is a great advantage in terms of the independence of user password encryption, and therefore the possibility of a revocation of an existing access password, on the other hand, it is a disadvantage of potential loss of data when the key header(of which we did not have a backup) is overwritten. LUKS header also ensures that a partition will not be seen as a file-system (ext2, vfat) and other. A plain

[4]PBKDF2 (Password-Based Key Derivation Function 2) is key derivation functions with a sliding computational cost, aimed to reduce the vulnerability of encrypted keys to brute force attacks.

dm-crypt partition may look like a unencrypted filesystem, and has a chance of being accidentally overwritten and destroy our data. LUKS improves ease of use and cryptographic security. We can easily change the password without having to re-encrypt the partition. Likewise, we can use multiple passwords to allow multiple users to access the encrypted disk/partition. LUKS is a standard for storing keys which are used by dmcrypt for encrypting the disk. Passphrases or key-files are hashed and stored on disk in the LUKS header in the encrypted partition.

- **Plain** mode does not store the type of encryption used, it requires supplying the same options each time (it does not contain LUKS header). *"In dm-crypt plain mode, there is no master-key on the device, so there is no need to set it up. Instead the encryption options to be employed are used directly to create the mapping between an encrypted disk and a named device. The mapping can be created against a partition or a full device [36]."* Plain mode has a lot of disadvantages. It does not provide the password management (single device has single password). These problems are solved by LUKS mode.

The syntax for using cryptsetup with LUKS extension:

```
cryptsetup -v luksFormat <device>
```

This is a command to create a new LUKS partition on device with default parameters. The cryptsetup action to set up a new encrypted device in LUKS mode is *luksFormat* which does not format the device, but sets up the LUKS header for selected device and encrypts the master-key with the desired cryptographic options.

Another LUKS actions are:

- *luksOpen* - opens the LUKS partition and sets the initial key
- *luksClose* - same as remove
- *luksAddKey* - adds key-file/passphrase to a LUKS partition
- *luksDump* - prints the information about the LUKS header

2.1.2.1 LUKS header

LUKS header stores cryptographic metadata and wrapped encryption key.

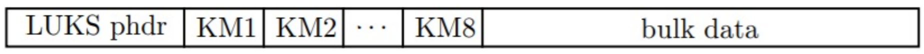


Figure 2.2: LUKS header [7]

A LUKS partition starts with the LUKS partition header (phdr). It contains information about the cipher name, cipher mode, the key length, a uuid, master key checksum. Following example is a sample of the *luksDump* command that lists the LUKS header content of the LUKS. Assuming, that `/dev/sdb2` is the LUKS enabled device:

```
cryptsetup luksDump /dev/sdb2
LUKS header information for /dev/sdb2

Version:        1
Cipher name:    aes
Cipher mode:    cbc-essiv:sha256
Hash spec:      sha1
Payload offset: 4096
MK bits:        256
MK digest:      e5 88 07 f2 4b ce 79 21 85 34 f7 a6
   ↪ e3 0b 6b b2 a7 b8 d5 a1
MK salt:        0c dd 95 3d 1e 30 1f 66 d4 5e 31 03
   ↪ 12 a0 61 29
                e5 ef 34 8e 13 5d 80 76 8b 4a 0a c3
                   ↪ 55 02 22 d3
MK iterations:  5750
UUID:           e4971160-047b-49ce-8246-b63f1fb67db9

Key Slot 0: ENABLED
        Iterations:             23233
        Salt:                   ff bc fc 78 98 5d 35
           ↪ 50 97 76 37 b4 70 99 38 44
                                9f bd a1 b9 02 2d 4d
                                   ↪ 1d 18 b5 dc f6
                                   ↪ 4c a0 37 fc
        Key material offset:    8
        AF stripes:             4000
Key Slot 1: DISABLED
Key Slot 2: DISABLED
Key Slot 3: DISABLED
Key Slot 4: DISABLED
Key Slot 5: DISABLED
Key Slot 6: DISABLED
Key Slot 7: DISABLED
```

LUKS header also contains an information about the key slots which stores up to 8 keys material from Key Slot 0 to 7. The bulk data is encrypted by the master key.

The phdr stores an information about every single key slot, which is associated with a key material section after the phdr. If the key slot is activated (ENABLED), it stores an encrypted copy of the master key in its key material section, which is then locked by user password which are we prompted to enter. Otherwise it is set do DISABLED. This user password unlocks the master key. The master key then can unlock the bulk data. The information how to decrypt a key material of a key slot with a given user password is stored in the phdr (e.g. salt, iteration depth). A LUKS partition can have up to 8 passwords, because the number of key slots is 8. To get to an encrypted partition, we can enter only one of these password [7].

2.1.3 Cryptsetup settings

This section is meant to provide an information to the individual cryptographic primitives used in dm-crypt/LUKS and how to set it up with the cryptsetup.

Cipher-name and chainmode must be compatible among different LUKS-based implementations. The user chooses an encryption algorithm with the *–cipher* parameter:

```
cryptsetup --cipher=<cipher name>-<chainmode>-<IV
   ↪ generator>
```

LUKS has to map them into something convenient to ensure, that the underlying cipher system can utilize the cipher name and chainmode strings, and these strings might not always be native to the cipher system.

Valid cipher names are AES, Twofish, Serpent, Cast5, Cast6. Valid chainmodes are ecb, cbc-plain, cbc-essiv:*hash*, xts-plain64.
Chainmodes using IVs (initialization vector) and tweaks must start from the all-zero IV/tweak. *"These IVs and tweaks cipher modes cut the cipher stream into independent blocks by reseeding tweaks or IVs at sector boundaries. The all-zero IV/tweak requirement for the first encrypted/decrypted block is equivalent to the requirement that the first block is defined to rest at sector 0 [7]."* Valid hashes are hash-spec, sha1, sha256, sha 512 and ripemd160. A compliant implementation does not have to support all cipher, chainmode or hash specifications.

Default mode is configurable during compilation, you can see compiled-in default options. When we enter command `cryptsetup` with the parameter `--help`, at the bottom is an information about Default compiled-in device cipher parameters.

```
cryptsetup -- help

LUKS1: aes-xts-plain64, Key: 256 bits,
LUKS header hashing: sha1, RNG: /dev/urandom
```

The cipher consists of three parts in format: cipher-chainmode-IV generator. The hash is used to create the key from the passphrase. To view full list of the supported ciphers and modes, we should check `/proc/crypto`. We can change it in command line using `<OPTIONS>` and select *–cipher* as described in section 2.1.2.

2.1.3.1 Ciphers

2.1.3.1.1 AES

The Advanced Encryption Standard [37], originally named Rijndael, is a symmetric block cipher that encrypts and decrypts with the same key data divided into blocks of fixed length. The cipher was published by Vincent Rijmen and Joan Daemen in 1998. AES cipher is fast in software and hardware [38] and, unlike its predecessor, DES does not use the Feistel network, but it consists of several rounds of substitution and permutation. It won its name in AES contest[5].

2.1.3.1.2 Twofish

Twofish [39] is a block cipher which was published by Bruce Schneier. Twofish is related to the earlier block cipher Blowfish. It uses a Feistel network and its S-boxes are key-dependent. Size of the key are most commonly implemented with key sizes of either 128, 192, or 256 bits.

2.1.3.1.3 Serpent

Serpent [40] is a symmetric key block cipher which was designed by Ross Anderson, Eli Biham, and Lars Knudsen. Size of the key are also most commonly implemented with key of either 128, 192, or 256 bits. It is an extensive to a DES. Serpent was designed for maximum security. Authors considered 16-round Serpent to be sufficiently secure against known types of attack, but the published version of Serpent used 32 rounds. Serpent was nearly unbreakable, but at the expense of its speed and it did not won AES contest but ended up second.

2.1.3.1.4 Cast5

CAST5 (alternatively CAST-128, defined in RFC 2144[6] in 1997) is a symmetric key block cipher. CAST5 is a 16-round Feistel cipher, with each round using 4 S-boxes with 8-bit input and 32-bit output. The algorithm was created in 1996 by Carlisle Adams and Stafford Tavares.

[5]The AES contest was organized by NIST and its goal was to choose a new block cipher and name it AES.

[6]`https://tools.ietf.org/html/rfc2144`

2.1.3.1.5 Cast6

CAST-256 uses the same elements as its predecessor CAST-128, including S-boxes, but is adapted for a block size of 128 bits – twice the size of its 64-bit CAST-128. CAST-256 is composed of 48 rounds Feistel cipher. It was not among the five AES finalists.

2.1.3.2 Initialization vector mode

Initialization vector (IV) is a random number which can be used with a secret key for encryption of data (message, plain text etc.). Using IV makes decryption much more difficult, because a sequence of an encrypted text might look similar, so the attacker could guess the corresponding text in the message using dictionary attack. The IV prevent the appearance of duplicate characters in encrypted text.

2.1.3.3 Block cipher mode of operation - chainmode

The purpose of cipher modes [41] is to mask patterns which exist in encrypted data. The long plaintext stream is divided into a series of blocks, and each cipher operates on these blocks one at a time.

2.1.3.3.1 ecb

Electronic Codebook (ECB) is the simplest block cipher mode of operation. The plain text stream is divided into blocks, and each block is encrypted independently following the same process with the same key. There are not used IVs.

2.1.3.3.2 cbc-plain

Cipher Block Chaining (CBC) was invented by Ehrsam, Meyer, Smith and Tuchman in 1976. In CBC mode, in compare to EBC, each block of plain text stream is XORed with the previous ciphertext block and then encrypted. Each block depends on all plaintext blocks processed up to that point. The CBC chaining is cut every sector, and reinitialized with the sector number as initial vector used in the first block. Plain is meant here to be an initialization vector mode *Plain-IV*, which initializes IV with the sector number converted to 32-bit version of the number encoded in little-endian padded with zeros as described with this formula:

$$IV(sector) = le32(sector)$$

So the whole *–cipher* option can be a combination of a block cipher mode and its IV mode. Or we can also add a hash as described below.

2.1.3.3.3 cbc-essiv:*hash*

Encrypted Salt-Sector IV (ESSIV) mode is used here for operating with cipher. It is also an initialization vector mode defined as:

$$IV(sector) = \varepsilon_{salt}(sector)$$

ESSIV has the IV from the key material due to an encryption of the sector number with a hashed version of the key material which is named a salt[7]. *Hash* is used for generating the IV key for the original key. For instance, when using sha512 as hash, the chainmode is *cbc-essiv:sha512*. Digest size of the hash must be a valid key size for the block cipher. For example, sha256 is a good choice for AES, because sha256 produces a 256-bit digest. This mode was developed for linux 2.6.10 to prevent watermarking attacks[8].

2.1.3.3.4 xts-plain64

XTS is for the purpose of disk encryption and it is recommended in an IEEE standard [43]. XTS is tweakable mode of operation and it is derived from XEX cipher. XEX stands for XOR Encrypt XOR. *Plain64* is 64-bit version of plain initial vector.

2.2 Trusted bootloader

Trusted bootloader is an extension of the bootloader with TCG support and one of the implementations is TrustedGRUB (Trusted Grand Unified Bootloader). To run TrustedGRUB, the system must include a TPM and a compatible TCG 1.1b BIOS. It is a free software under the GNU GPL (GNU General Public Licence). After the CRTM measures the BIOS and BIOS measures additional ROMs and stores them in specified PCRs, bootloader comes in a row and completes an idea of chain of trust.

GRUB is able to boot more OS so we can use any including Linux as well as other Unix-based platforms and Microsoft Windows without need to specify the physical address of the kernel on the disk. It also supports the multiboot. The path to the kernel can be given by specifying a name of the file, drive and partition in which kernel is stored.

GRUB consists of multiple stages. The first stage, Stage 1, is located in the boot sector or in the MBR. Its size is always 512 bytes. Next stage can be located on different media (e.g. a hard disk or floppy), so Stage 1 have

[7]It is a misapply of the term “salt”, because a salt is usually generated randomly. Here it is deduced from key material.

[8]Watermarking attack is an attack on disk encryption methods where the presence of a specially crafted piece of data can be detected by an attacker without knowing the encryption key [42].

to support addressing modes - two for hard drives (Logical Block Addressing (LBA) and Cylinder Head Sector (CHS)) and one for floppy disks.

If you boot from a hard drive, Stage 1.5 will be used. Stage 1 measures the first sector of Stage 1.5. The first sector of Stage 1.5 measures the remaining sectors. After loading the remaining sectors, Stage 1.5 measures stage 2. Stage 2 is the main part of GRUB. Its task is to arrange that the operating system kernel can be loaded. The whole booting process with TrustedGRUB is illustrated in Fig. 2.3.

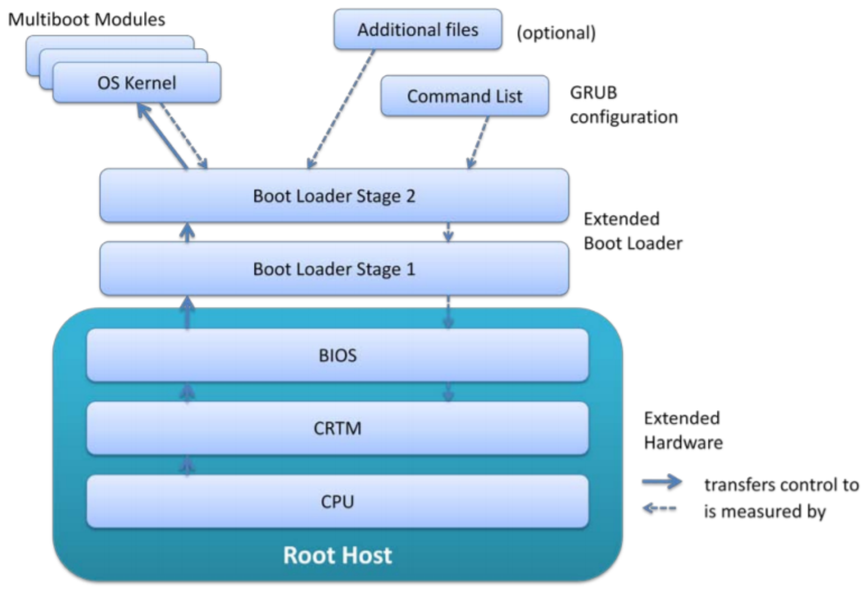


Figure 2.3: Booting with TrustedGRUB [8]

After successful deployment, TrustedGRUB measures the grub configuration file and stores it into PCR 5.
TrustegGRUB has an additional functionality called ”checkfile -option” where verifies given files by comparing SHA-1 results in PCR with checkfile (which was precalculated) and store its values into PCR 13. The files to be measured with SHA-1 hash of the succeeding file and given path of file in the option checkfile [8]. The check file itself contains a list of tuples of arbitrary length and must not be larger than 8096 Bytes. Its syntax is as follows:

```
2647eeae7290c5a58dacb87347ba074de7e47bac (hd0,1)/etc/
    ↪ passwd
a97fbdba48d4a6340baff683941079dde56044e0 (hd0,1)/etc/
    ↪ shadow
6fc01c858d17593a309b91d5fe5859c545409861 (hd0,0)/home
    ↪ /test.sh
```

The first value is a 40 byte SHA1 hash value of the succeeding file, followed by a single white space character. The second component is absolute path together with the drive reference of the file corresponding to the hash value, followed by a new line character. Syntax must be correct, otherwise TrustedGRUB is not able to boot. The integrity of all files listed in this check file is checked when your system is booted. It compares the referenced hash values to newly computed ones. If values do not match, our option is either continue booting into a untrusted system or stop the boot process.

2.3 Tools for communication with the TPM

To communicate with the TPM, tools tpm_tools and TrouSerS, needs to be installed. A brief description of tools used for my thesis is presented in this section.

2.3.1 TrouSerS

TrouSerS is an open-source implementation of TSS Specification 1.2 (see section 1.2.7). The latest version is 0.3.14 by [44]. The driver has been designed to meet TIS specification (TPM Interface Specification) and supports TPM from various manufacturers - Atmel, Infineon, STM. According to specification 1.2 [45], this interface has been standardized for every manufacturer. This module is called `tpm_tis`. All parts are included in the `trousers` package. It has also several component:

TCS daemon(tcsd) is running only in a user mode. According the TSS specification, only tcsd have access to the TPM device driver. *"At boot time, tcsd should be started, it should open the TPM device driver and from that point on, all requests to the TPM should go through the TSS stack. The tcsd manages TPM resources and handles requests from TSP's both local and remote [46]."*

The TSP shared library allows communication between applications and tcsd locally or remotely. The TSP controls the resources used for their communication.

Persistent Storage (PS) files - can be a database for keys with each key indexed by UUID (Universal Unique Identifier). It exists two different kind of PS for keys:

- User PS is maintained by the shared TSP library. Each user will have their own custom PS, which is stored in the `~/.trousers/user.data` file. If we want the TSP to use a different file for the custom PS, set the path to it using the TSS_USER_PS_FILE environment variable. This file will be created when the first application key is saved.

- System PS is maintained by TCS. The saved keys are available even after tcsd has been rebooted or system reset until we explicitly request removal.

  In `usr/local/var/lib/tpm/system.data` is the System PS file by default. This file is created by taking ownership of the TPM.

A config file (tcsd.conf) - Located in `/usr/local/etc/tcsd.conf` by default [46].

2.3.2 tpm-tools

Tpm-tools is an open-source package that allows users and applications to use the TPM. The latest version is 1.3.9.1 in 2017 [47, in Files]. The package also contains commands to use some of the available features in the TPM PKCS#11 interface implemented in OpenCryptoki [47]. If this is not the case, tpm-tools will only contain TPM commands after the compilation.

2.3.2.1 TPM commands

To communicate and perform a command with the TPM, the tpm-tools provides the list[9] of commands. Here are one of the most used:

tpm_takeownership sets the TPM owner. The command asks for new password for SRK and TPM owner, and then asks for confirmation.

tpm_changeownerauth changes authorization credentials for SRK and TPM owner. The command prompts you to enter the current and new password and asks for a new password.

tpm_clear transfers TPM to ”Unowned”, ”Disabled”, and ”Inactive”. This command clears all the information associated with the TPM owner.

tpm_createek creates an EK. This command is rarely used, because in most cases the EK is generated during TPM production.

tpm_getpubek will show the public section of the EK. This operation may be limited and require TPM owner authorization.

tpm_restrictpubek only the TPM owner can read the public part of the EK. To successfully complete this command, we need to prove that we know the TPM owner's secret.

tpm_selftest asks the TPM for self-testing. Its result will be printed. If the self-test does not succeed, the TPM goes into a failure mode in which no further command is received. **tpm_setactive** changes the state of TPM from ”active” to ”inactive” or vice versa. The command requires authorization of the TPM owner.

tpm_setclearable disables the TPM clear operation.

[9] `https://www.mankier.com/package/tpm-tools`

tpm_setenable changes the TPM status from ”accessible” to ”disabled” or vice versa. The command requires authorization of the TPM owner.

tpm_version prints the manufacturer and the TPM version.

tpm_sealdata seals sensitive input data to the SRK of the system’s TPM and optionally a PCR configuration.

tpm_unsealdata the opposite function to the seal function.

2.3.2.2 TPM commands which access to NVRAM

NVRAM is a storage area inside TPM. It has only 1280 bytes. They are controlled by an owner but permission can be delegated. The full list of permissions is described in [48].
There are plenty of way to use NVRAM [49].

- Storage for data that can serve as reference
  - It is harder to modify than data on disk
  - Hashes for integrity checking
- Storage for high-value data
  - Keys
  - Certificates

Commands for work with NVRAM:

tpm_nvdefine defines a new NVRAM area at the given index and of given size. Permissions bits that control access to the NVRAM area must be set by user. One of the permission must be a protection on write to NV area otherwise will not be created.

tpm_nvinfo displays information about defined NVRAM areas using parameters -i to display NVRAM area with the given index. With parameter -n, it displays only available NVRAM areas indices.

tpm_nvrelease releases an NVRAM area of a given index if it is used. If owner authentication is required then the user must provide the owner password.

tpm_nvwrite writes data to an NVRAM area. There are two main parametres - the index of NVRAM area and the data to write.

tpm_nvread reads data from the NVRAM area and either displays them on a command line with the parameter -i (index) or writes them into a file with the -f parameter.

More information can be found on a manpage [50].

2.3.3 Initialization of the TPM

`Tpm_init` command is a method of initialising the TPM and puts it into a state where it waits for the `tpm_startup` command. `TPM_init` is a physical method of initializing a TPM. On a PC this command arrives to the TPM via the bus (LPC, SMBus) and informs the TPM that the platfom is performing a boot process so that is not a command that any software can execute.

2.3.3.1 Tpm_startup command

The type of startup mode of TPM is divided into these three modes:

- Clear - results in the TPM values are being set to a default state.
- Save - TPM recovers saved state, including PCR values, with the execution of the `tpm_SaveState` command.
- Deactivated - state, which informs a TPM than any further operations should not be allowed. From now on, TPM can be only reset by executing `tpm_init` command again.

According to [51] not all TPM functions are available until the TPM acquires an owner.

2.3.4 Enabling the TPM chip

There are three discrete states of the TPM - enabled/disabled, active/inactive and owned/unowned. These stated form eight operational modes ilustrated in Fig. 2.4.

The TPM chip is delivered on the motherboard in the operational mode S8 which is `Disabled`, `Inactive` and `Unowned` state. The most important operation is to enable the TPM chip in BIOS for initiating a communication. If we do not enable it in BIOS, it will not be visible and accessible.

The TPM chip never starts a communication itself, only responds to the commands that comes from above.

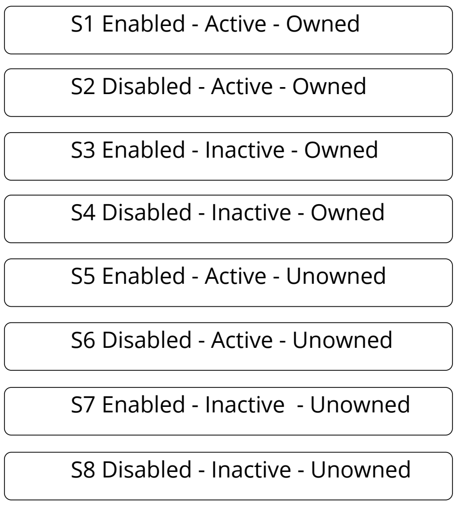

S1 Enabled - Active - Owned
S2 Disabled - Active - Owned
S3 Enabled - Inactive - Owned
S4 Disabled - Inactive - Owned
S5 Enabled - Active - Unowned
S6 Disabled - Active - Unowned
S7 Enabled - Inactive - Unowned
S8 Disabled - Inactive - Unowned

Figure 2.4: Operational modes [9]

To clarify what some of the mode can be useful for, here are some examples from practice as written in a specification [9]: *"Mode S5 can be used for when a corporate customer wishes to have platforms shipped to their employees and the IT department wishes to take control of the TPM remotely. When the platform connects to the corporate LAN the IT department would execute the TPM_TakeOwnerShip."*

S1 mode if mode where all TPM functions are available. S8 mode is a mode where all TPM functions are off.

Practical part will contain a demonstration of how to get to mode 1 from mode 8 and 7.

2.4 Conclusion

Dm-crypt along with its superstructure LUKS offers a great deal of possibilities to encrypt our disk, simply by choosing the right cryptographic parameter option to encrypt our data. It is also only one cryptographic tool that is compatible with the TPM under the Linux. Default mode is *aes-xts-plain64*. These cryptographic functions were chosen because they are among the ones most commonly used in the disk-encryption and they are sufficient to fulfill most requirements.

Security is also increased by the fact that the TCG has introduced a limitation that only TCS daemon can communicate with TPM via TDDL. To run this driver (see 1.2.7.1.1), we need a module that runs it. The module TIS was standardized for all manufacturers, however not all PCs loads this kernel module automatically. In practical part I will explain how to load it manually.

Chapter 3

Practical part - Realization

This chapter deals with the USB mass storage device encryption and demonstrates the entire effect of changes in the platform on the access to the secured disk using TrustedBoot. It serves as a guide of how to set up the TPM for testing one specific function - sealing.

For ease of use, encryption of the USB mass storage device (USB disk) has been selected. USB disk encryption is more practical and safer for trying out TPM chip manipulation. This whole section contains a sample code and simple commands which encrypt only one USB disk with one partition. This is simply for the readers to better understand simple issues. Whole bash script with more features is attached to this work.

More used method of an encryption is a full disk encryption, where we can encrypt the entire root file-system. Unfortunately, this is not good method for a demonstration, but in practice, we would use this method. One of the possible guide is here [52].

The main features of the TPM chip are the storage of cryptographic fingerprints for hardware and software during booting and the verification if we change the integrity of the measured platform. The TPM refuses to release encrypted files without a password because it boots into an untrusted state.

Everything was written under the Linux. I used Bash Programming Language for the implementation.

In the practical demonstration, I will assume, that we have an USB storage media and its device name is `/dev/sdb`.

3.1 Platform used

Lenovo ThinkPad x61s computer with operation system Linux Mint version 18.01 with TPM chip version 1.2 was available for a practical demonstration. The solution has been adapted to the possibilities of these versions. Solution may differ, because of the various versions of Linux. The encryption of data on the external hard drive itself has been used with the Linux Unified Key Setup (LUKS).

3.2 Detecting the presence of the TPM chip

In order to test TPM commands, it must be part of the computer's motherboard. The best indicator of its presence is the options in the BIOS. The chip version was TPM 1.2. Its presence is indicated by the BIOS under the `Security Chip` which contains these settings:

- `Disabled` - Security chip is hidden and disabled.
- `Active` - The security chip is visible and functional.
- `Inactive` - The security chip is visible but nonfunctional.

3.3 TPM chip initialization

The chip was in the `Disabled` state. I had to make it available in BIOS. The platform I was accessing contained an option `Clear` in the BIOS, which was only available when the computer was shut down and then turned on, not after the reboot. This setting is equivalent to the `tpm_clear` command from the tpm-tools package (see section 2.3.2).

3.4 Enabling the TPM

In my case, I had to enable it by setting `Active` option. Different platforms use different methodes.

If the TPM chip was detected, its driver reported in dmesg:

```
dmesg | grep -i tpm
[    2.616330] tpm_tis 00:05: 1.2 TPM (device-id 0
   ↪ x3203, rev-id 9)
```

where TIS is the kernel module (driver) standardized for every manufacturer of TPM. If kernel module is not loaded, we can load it manually by typing:

```
modprobe tpm_tis
```

and check *dmesg* output again.

3.5 Installing necessary packages

When working with the TPM chip, packages tpm-tools and TrouSerS needs to be installed. Commands:

```
sudo -i
sudo aptitude install tpm-tools trousers
```

To determine the TCG version, type the following command:

```
cat /sys/devices/pnp0/00\:0*/caps
Manufacturer:
TCG version: 1.2
Firmware version: 13.9
```

The above output shows that the TPM/TCG chip version is 1.2.

3.6 Taking ownership

The `tpm_takeownership` command (see section 1.2.3) sets the TPM owner. It creates a 2048-bit RSA SRK and then asks the owner to confirm it. It initiates the following functions: stores the owner password, then stores the SRK and returns the public part of the SRK to the owner. With a functional SRK, we now have a functional TPM and we are able to create and use signing and encryption keys.

```
tpm_takeownership -z
Enter owner password:
Confirm password:
```

The `-z` parameter set a secret of all zeros (20 bytes of zeros) automatically as the SRK secret. This parameter is used for easier demonstration, because we will not be able to ask for SRK later, when manipulating with the tpm commands. For higher safety we would type another SRK password.

3.7 Use of PCR registers

The practical part uses the features of the PCR registers and its *sealing* method (see section 1.2.6.2). When all the necessary packages are installed, enabled TPM chip in BIOS, the contents of the PCR registers can be seen in the `/sys/class/tpm/tpm0/device/pcrs` folder for Kernel versions older than 4.x or in the `/sys/class/misc/tpm0/device/pcrs` folder for Kernel versions younger than 4.x. Their content looks like this:

```
cat /sys/class/tpm/tpm0/device/pcrs
```

```
PCR-00: 70 5F 0A 19 BE E6 87 D7 3B 1D 5B 28 44 12 B3 2C 88 10 6B 9F
PCR-01: 48 DF F4 FB F3 A3 4D 56 A0 8D FC 15 04 A3 A9 D7 07 67 8F F7
PCR-02: 53 DE 58 4D CE F0 3F 6A 7D AC 1A 24 0A 83 58 93 89 6F 21 8D
PCR-03: 3A 3F 78 0F 11 A4 B4 99 69 FC AA 80 CD 6E 39 57 C3 3B 22 75
PCR-04: 7C B2 10 EC 89 79 EE 60 3A 21 9B 92 86 4C 11 E5 5F 9A 59 FD
PCR-05: 99 F4 50 AB 2B 15 13 C7 42 05 2E 83 13 A8 13 67 4A 6E 90 A8
PCR-06: 58 5E 57 9E 48 99 7F EE 8E FD 20 83 0C 6A 84 1E B3 53 C6 28
PCR-07: 3A 3F 78 0F 11 A4 B4 99 69 FC AA 80 CD 6E 39 57 C3 3B 22 75
```

```
PCR-08: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-09: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-10: F3 1E 9C 77 9D C3 AB E0 49 7A C1 D1 46 25 CE 03 FD 0C 55 4E
PCR-11: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-12: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-13: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-14: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-15: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-16: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-17: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-18: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-19: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-20: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-21: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-22: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-23: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

This is how PCR's values looks like only with SRTM but without a TrustedGRUB2 and without DRTM as mentioned in section 1.2.5.3.2.

The first 0-7 results of PCR registers are already stored together with PCR 10, which is used for kernel and initrd measurements. To get a secure boot, we need to install TrustedGRUB2 as mentioned in section 2.2.

3.7.1 Installing TrustedGRUB version 2 - TrustedGRUB2

As is written at the download site, version 2 was highly inspired by the former projects TrustedGRUB and GRUB-IMA. However TrustedGRUB2 was completely written from scratch. To perform a secure boot, TrustedGRUB2[10] need to be installed and following commands taken from the download site should be performed:

```
sudo aptitude install autogen autoconf automake gcc
   ↪ bison flex
./autogen.sh
./configure --prefix=`pwd` --target=i386 -with-
   ↪ platform=pc
make
sudo make install
sudo ./sbin/grub-install --directory=`pwd`/lib/grub/
   ↪ i386-pc /dev/sd<x>
```

After reboot, all PCR, except for PCR 12, are filled with measurements. PCR 12 is separated for LUKS header, which we will be using later while encrypting device. After TrustedGRUB2 being installed, we should reboot system and check our PCR values again.

```
cat /sys/class/tpm/tpm0/device/pcrs
```

[10]file can be downloaded here: `https://github.com/Rohde-Schwarz-Cybersecurity/TrustedGRUB2` (last visit 24.7. 2017)

```
PCR-00: 70 5F 0A 19 BE E6 87 D7 3B 1D 5B 28 44 12 B3 2C 88 10 6B 9F
PCR-01: 48 DF F4 FB F3 A3 4D 56 A0 8D FC 15 04 A3 A9 D7 07 67 8F F7
PCR-02: 53 DE 58 4D CE F0 3F 6A 7D AC 1A 24 0A 83 58 93 89 6F 21 8D
PCR-03: 3A 3F 78 0F 11 A4 B4 99 69 FC AA 80 CD 6E 39 57 C3 3B 22 75
PCR-04: 37 F9 1C 96 A4 A0 D2 40 B1 50 4A 35 86 CF D5 66 50 59 11 4E
PCR-05: 56 CF F3 16 CA 86 D2 9D 1F 7B EA E7 35 E9 4F 3C 77 4F C5 DD
PCR-06: 58 5E 57 9E 48 99 7F EE 8E FD 20 83 0C 6A 84 1E B3 53 C6 28
PCR-07: 3A 3F 78 0F 11 A4 B4 99 69 FC AA 80 CD 6E 39 57 C3 3B 22 75
PCR-08: D3 F6 C9 85 14 27 D4 09 F4 77 F9 F4 98 DD C3 5B 3C 7A 84 E4
PCR-09: B3 3A CD A8 64 4D 78 1C E4 51 A0 C0 7D DA F2 F8 15 41 A8 3C
PCR-10: AB CC 52 78 A8 F9 48 17 25 15 47 32 39 BA 7E 34 2C D7 69 B3
PCR-11: 19 6F 30 0F 19 B0 12 C7 B7 C5 20 24 3D 57 DC 74 7C C3 92 F8
PCR-12: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-13: A2 62 C2 BE 50 B6 89 AF F8 BD 72 67 08 58 A7 75 C3 F0 9C 8C
PCR-14: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-15: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-16: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR-17: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-18: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-19: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-20: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-21: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-22: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-23: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

The PCR 8, which measures the first sector of TrustedGRUB2 kernel (diskboot.img), the PCR 9 which measures TrustedGRUB2 kernel (core.img), the PCR 11 which measures all command line arguments from scripts (grub.cfg) and those entered in the shell and the PCR 13, which measures parts of GRUB2 that are loaded from disk like GRUB2-modules are now filled up with values (taken from the TrustedGRUB2 readme [53]). PCR 0-7 are the same. PCR 10 was filled with a value before, and now it is again, but with different value. This is because the component measures another component and puts this measurement into the PCR before passing it through with the operation *extend* (see 1.2.4). Values are expanded by a new value using the previous value (hash) due to Chain of Trust.

3.7.2 Creating a key

For a key material, we can create either a key-file, or enter a passphrase in command cryptsetup. Cryptographically the safest, I chose a key-file named `secret.bin` as a key material, using `/dev/random` to create a key. `/Dev/random` is a special file in the Unix systems, which is interface for an output of PRNG[11]. In Linux, the generator is implemented inside the kernel, where entropy is also collected and this interface is called *blocking*, which basically means that in the absence of entropy it waits until it is enough entropy again and the calling program does not continue until then. The *non-blocking* variant is then `/dev/urandom` when a ”less random number” is returned in the absence of entropy. In other systems, implementation may be different and it is inappropriate for cryptographic material for key creation, so there is a

[11] Pseudo-random number generator (PRNG)

need to make random and not urandom. `/Dev/random` collects data from the outside of the system, e.g. mouse moves, typing on keyboard, arrival times, anything that can not be estimated in advance, some uncertainty. After, we will add it to our keys for the LUKS-encryption partition and store this key in the TPMs NVRAM (non-volatile RAM) to use for decryption.

```
sudo dd bs=1 count=32 if=/dev/random of=/dev/shm/
    ↪ secret.bin
chmod 700 /dev/shm/secret.bin
```

The device `/dev/shm` is a RAM disk device. It is good to have our key in volatile memory to increase security. Maximum of the key-file size can be up to 8192 kiB.

3.7.3 Creating a LUKS partition

To create a LUKS partition, we need to create a partition on the `/dev/sdb`. In order to use the device with LUKS options, LUKS header must be added using the *luksFormat* command [2.1.2]. It will initialize a LUKS partition and set the initial passphrase, and encrypt the master-key with the desired cryptographic option. Default is aes-xts-plain64. First key-slot will be used. It is necessary to backup the device (because LUKS device initialization erases old data), do a *luksFormat*, and restore our backup on the now encrypted device. Assuming, that `/dev/sdb1` is our new partition, we can accomplish it by typing:

```
sudo cryptsetup luksFormat /dev/sdb1
WARNING!
========
This will overwrite data on /dev/sdb1 irrevocably.

Are you sure? (Type uppercase yes): YES
Enter passphrase:
Verify passphrase:
```

When typing *luksFormat*, the device now has a LUKS header which contains the type of cipher which was used to encryption. The passphrase used to encrypt LUKS device is stored to the Key-slot 0.

3.7.4 Adding a key file to LUKS

LUKS devices may hold up to 8 different key-files/passwords. This key-file is an additional authorization method. When adding new key-slot to LUKS device, it asks us for a passphrase, and we type a new passphrase, but then complains that there is no key slot with that passphrase. That is what it is intended. We are asked a passphrase of an existing key-slot first, before we can enter the new passphrase for the new key-slot. Otherwise we could break the encryption by just adding a new key-slot. This way, you need to know the passphrase of one of the already configured key slots to configure a new key slot. With known passphrase and parameter –key-file to `luksAddkey`, we can store our secret key-file to another key-slot.

```
sudo cryptsetup luksAddKey /dev/sdb1 /dev/shm/secret.
    ↪ bin
```

To see filled key-slots, just type:

```
sudo cryptsetup luksDump /dev/sdb1
```

It will list the LUKS header of the device with the cryptographic option used and the two filled key-slots.

3.7.5 Storing the secret directly into the TPM

Commands that defines new NVRAM area [10]:

```
# INDEX of NVRAM in which we want to store our secret
# Atmel TPM threw errors if NV index 1 was used
INDEX=2
# PCRS of which we want to seal the data to
PCRS="-r0 -r1 -r2 -r3 -r4 -r5 -r6 -r7 -r8 -r9 -r10 -
    ↪ r11 -r12 -r13"
# Creating an index of NVRAM area of given size and
    ↪ permissions
tpm_nvdefine  -i $INDEX  -s $(stat -c "%s" /dev/shm/
    ↪ secret.bin) -p "OWNERWRITE|OWNERREAD" -z "$PCRS
    ↪ "
```

First, we have to define a new NVRAM where we can keep a key in which is sealed to certain PCRs, We have to create index using the `tpm_nvdefine`. The parameter -i stands for the number of index, parameter -s is the size of NVRAM area. The parameter -z uses a secret of all zeros (20 bytes of zeros) as the owner's secret. The parameter -r (the most important) stands for PCRs to seal the NVRAM area to for reading (used multiple times in a variable PCRS).

I have set the permission of the NVRAM - OWNERWRITE and OWNERREAD. Both permission reading/writing requires owner authorization. Others may be more appropriate, since it depends on our situation (see 2.3.2.2).

Then, if creating an index succeeded, writing our key into the NVRAM is needed.

```
tpm_nvwrite -i $INDEX -f /dev/shm/secret.bin -z
```

3.7.6 Filesystem

We need to unlock the encrypted partition, map it to `/dev/mapper` and build a filesystem.

```
cryptsetup luksOpen /dev/sdb1 <dmname>
mkfs.<type>/dev/mapper/<dmname>
```

Using `<type>` as a `vfat`[12] filesystem. When we create mapping, we should format the filesystem in order to be able to read the content of our device-mapper. But we should format it only once, otherwise, all our data will be overwritten by a new formatting.

3.7.7 Making shell script executable after USB connection

Since we want USB to be immediately unlocked when we connect it in the port, we need to write a rule in udev[13]: *"udev loads kernel modules by utilizing coding parallelism to provide a potential performance advantage versus loading these modules serially. The modules are therefore loaded asynchronously. The inherent disadvantage of this method is that udev does not always load modules in the same order on each boot. If the machine has multiple block devices, this may manifest itself in the form of device nodes changing designations randomly [55]."* So, if the machine has two hard drives, /dev/sda may randomly become /dev/sdb. And also when connecting a USB, it may occure as a /dev/sd[b-z]. If we want to write an udev rule for executing a shell script, it is necessary to write it into `/etc/udev/rules.d/<x>.my-rule.rules`, where `<x>` is the number of rule. Rules are executed according to the numbers in the order. My rule is:

```
ACTION=="add",
SUBSYSTEM=="block",
KERNEL=="sd[b-z][1-9]",
ENV{ID_FS_TYPE}=="crypto_LUKS",
RUN+="/usr/bin/tpm-luks-encrypt.sh"
```

Where `ACTION` is used when external programs runs upon certain events. `ACTION` environment variable detects whether the device is being connected or disconnected - `ACTION` can be either "add" or "remove". If we want to identify devices based on advanced properties such as vendor codes, product numbers, serial numbers, storage capacities, number of partitions, etc., the drivers export these information to sysfs[14] and udev allows us to incorporate

[12]https://www.powerdatarecovery.com/data-recovery/vfat.html

[13]Udev [54] is in Linux kernels 2.6 and provides a userspace solution for a dynamic /dev directory, with persistent device naming. Udev also handles all user space events raised while hardware devices are added or removed from the system.

[14] sysfs is file-system managed by the kernel, exports basic information about the devices currently plugged into system. udev can use this information to create device nodes corresponding to your hardware [54].

sysfs-matching into our rules, using the ATTR key with a slightly different syntax. `SUBSYSTEM` is written as a block, because of the sysfs tree hierarchy. It match against the subsystem of the device, or the subsystem of any of the parent devices. `KERNEL` match against the kernel name for the device. So every USB connected will be matched as a block in a sysfs tree hierarchy and kernel name for device will be `/dev/sd<x>`. We can check the all attributes and environments variable using `udevadm -q all -n /dev/sd<x>`. `ENV` is used for enviromental variables. Since sysfs exports basic information, udev use these information. Some of them are stored in environmental variables. It comes in handy e.g. when we want to know which USB device was plugged-in using environmental variable `DEVNAME` or if we want to know which type of file system is on partition using `ID_FS_TYPE`. I wanted to execute every partition of the USB which has a file system type matched as `crypto_LUKS` which is the result of the *luksFormat* command to avoid later control whether it is a valid LUKS format or not. I used `RUN` option to run my script written below.

```
#!/bin/bash

logger "Script RUNS"
MAPPER="map_secure"
INDEX=2
OUTKEYFILE="/dev/shm/tpm_temp$$.key"
DEVDEVICE="$DEVNAME"
```

The logger is for logging and detecting the USB when is connected and to control it in the log file in `/var/log/syslog`. MAPPER is to set device mapper (see 2.1.1) because encrypted data needs to be encrypted by mapping partitions to a new device name using the device mapper which will be stored in `/dev/mapper/<dmname>`. INDEX is the index of the TPM NVRAM where the secret is stored. OUTKEYFILE is the name of the file, where will be the key-file from the TPM temporary stored. DEVNAME is the environmental variable and contains the name of the device. This is a demonstration of encrypting one partition of one USB storage media, because the number of index of NVRAM is fixed. In my work, multiple partitions and multiple USB can be encrypted, which I will explain in a chapter 4.

```
#read key from TPM and copy to ram disk
tpm_nvread -i "$INDEX" -f "$OUTKEYFILE" -z
if [[ "$?" -ne 0 ]];then
    #Unable to unseal
    exit 1
fi
```

In order to get a content of the NVRAM out, the command `tpm_nvread` should be used. It must produce the same output as the

```
sudo hexdump -C /dev/shm/secret.bin
```

which is our secret key that we stored in a TPM NVRAM. If we want to read our encrypted USB disk, we should decrypt it first using `luksOpen` parameter which opens (creates a mapping with) `<name>` backed by device `<device>`.

```
# decrypt drive
cryptsetup luksOpen "$DEVDEVICE" "$MAPPER" --key-file
    ↪ "$OUTKEYFILE"

# zeros out the key in memory
SIZE=$(stat -c "%s" "$OUTKEYFILE")
sudo dd if=/dev/zero of="$OUTKEYFILE" bs=1c count="
    ↪ $SIZE"
# remove key from ram disk
sudo rm -f "$OUTKEYFILE"
```

While disconnecting an USB, another rule in udev should be written for unmounting a device mapper and closing a LUKS encrypted partition:

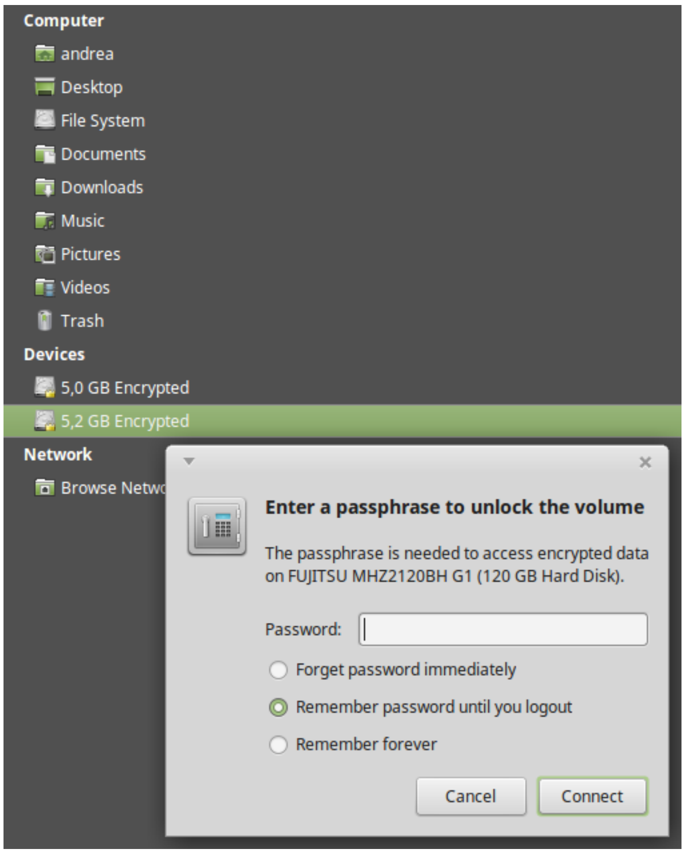
```
ACTION=="remove",
SUBSYSTEM=="block",
KERNEL=="sd[b-z][1-9]",
RUN+="/bin/umount  /dev/mapper/<dmname>"

ACTION=="remove",
SUBSYSTEM=="block",
KERNEL=="sd[b-z][1-9]",
RUN+="/sbin/cryptsetup luksClose  <dmname>"
```

3.7.8 Sealing the NVRAM

To seal (see 1.2.6.2) our configuration, meaning the NVRAM index can not be read if something is changed in the boot process (kernel, initrd, grub-modules, grub-arguments, etc...), we have typed all PCRs (see 3.7.5), as the `-r` parameter to a `tpm_nvdefine` command. The rest of PCR remain unchanged because they are used for other purposes, for example for DRTM (see 1.2.5.3.2). Now the system will not give out the keys from TPM NVRAM when something changes in a booting.

I saved the output of the PCRs before the reboot, then I typed `E` in a TrustedGRUB bootloader and after I wrote `echo "Hello World"` in a kernel command line and pressed `F10` to boot. We need to have GRUB interactive. My system did not give me an access to USB without typing a correct passphrase.



Enter a passphrase to unlock the volume

The passphrase is needed to access encrypted data on FUJITSU MHZ2120BH G1 (120 GB Hard Disk).

Password:

( ) Forget password immediately
[✓ Remember password until you logout]
( ) Remember forever

Figure 3.1: Screenshot of unlocking the volume

Then I typed the `diff` of this two outputs of measured PCRs and this is how it looked like:

```
diff before_reboot after_reboot
```

```
PCR-11: BA B5 DA 9D 5C D7 2A 4A 35 74 D8 BC 9B DB 0A 58 DE 71 38 50
PCR-12: BA 69 A2 D7 70 0A 78 EE 57 89 AA D0 3C 77 48 A1 0F FD 8D 34
PCR-13: A2 62 C2 BE 50 B6 89 AF F8 BD 72 67 08 58 A7 75 C3 F0 9C 8C
--
PCR-11: 9E 1F 99 C5 9E B9 A7 2E 84 16 2A 10 B5 6F 80 98 57 D0 27 A0
PCR-12: 66 64 06 DD 3A 3F 78 0F 11 A4 B4 83 58 93 89 6F B2 92 AC 06
PCR-13: CB 83 03 61 AE EE 55 1D C2 44 EB 52 6D 98 B6 91 81 AE 49 5D
```

As we can see, the measurement has changed from PCR 11, because it contains measurement for command line changes and after reboot, kernel command line measurements were changed because of typing `echo "Hello World"` in a TrustedGRUB (that was not measured before). This means that system refused to give out the data from the USB without typing a correct passphrase, because of the booting to an untrusted state. Another indication, that TPM will not let me read the TPM NVRAM value was, when I typed:

```
tpm_nvread -i 2
Enter NVRAM access password:
Tspi_NV_ReadValue failed: 0x00000018 - layer=tpm,
   ↪ code=0018 (24), Wrong PCR value
```

3.7.9 System update

Previous section was a demonstration of a system change by a non-privileged user - making changes in kernel command-line. A privileged user may also be denied access to TPM NVRAM. This happens especially during a system update. The whole measurements of the previous system will be changed and access and sealed keys to the TPM NVRAM will be lost. This was resolved by tpm-luks[15] and its script tpm-luks-update which was tested on RHEL 6 and Fedora 17.

When a new initramdisk[16] is generated by system, a hook which is stored in a

```
/etc/initramfs/post-update.d/tpm-luks-update
```

is executed. Tpm-luks-update (executed by a hook) then precomputes PCR values for the new system state and migrates the current LUKS key to a new NVRAM index in TPM. Basically, it searches for all used NV indexes. Then creates a temporary file on the ram disk, where the TPM NVRAM keys will be stored (of course, the user must provide himself by an owner password). The algorithm searches for new NV indexes to store the secret in. The new system state is precomputed and stored as a permission file, which will be then created with the `tpm_nvdefine` command with the parameter -f (-f, –filename, file containing PCR info for the NVRAM area, see manual [48]) defines a new NVRAM area with new index. At the end, the content in the old NVRAM index will be written into a new NVRAM index. This allows for easy kernel updates without need for a manual intervention. Since tpm-luks was implemented for RHEL 6 and Fedora 17, I did not tested the system update, but I will mention this issue for further work for Ubuntu. More information about this issue, with the script `tpm-luks-update` is here [56].

[15] https://github.com/shpedoikal/tpm-luks

[16] Initramdisk to which a default Grub points at the moment of execution of the tpm-luks-update

Chapter 4

Developed scripts

Two simple scripts were developed and are attached to this thesis. They are meant to be useful for users who would like to learn more about this issue and would like to secure their external drive using TPM and LUKS extension. I have expanded their functionality by creating a script that saves multiple keys to NVRAM at one time and decrypts them concurrently. This means we can have more encrypted partitions on one USB disk. Therefore it may not be just a single primary partition encrypted, but there may be more. It also extends a functionality, that more USB disks can be encrypted and decrypted at the same time. Before using these scripts, many manual installations, as described in chapter 3, have to be completed.

The first script, `tpm-luks-encrypt.sh`, is interactive and asks us for a partition which we want do encrypt. The list of all block devices can be found by command `lsblk`. Then it formats partition using *luksFormat* parameter. We can now choose the index where we want to store our secret key. I have limited it to the interval of index 2 to index 128 because of the lack of memory of the NVRAM. I have used the UUID (size of 36B) of the LUKS encrypted partition as a key in ”associative array” where is stored the key which encrypts a given partition.

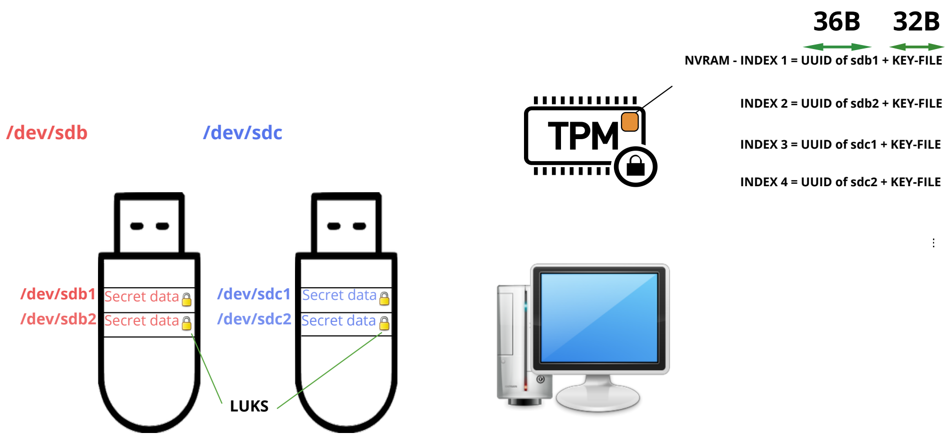

36B
32B
NVRAM - INDEX 1 = UUID of sdb1 + KEY-FILE
INDEX 2 = UUID of sdb2 + KEY-FILE
INDEX 3 = UUID of sdc1 + KEY-FILE
INDEX 4 = UUID of sdc2 + KEY-FILE
TPM
/dev/sdb
/dev/sdc
/dev/sdb1
/dev/sdb2
Secret data
Secret data
/dev/sdc1
/dev/sdc2
Secret data
Secret data
LUKS

Figure 4.1: Storing indexes into NVRAM

The second script, `tpm-crypto-mount.sh`, is executed, when USB is attached using udev rule and gets out the key for decrypting all LUKS encrypted partitions.

`tpm-luks-encrypt.sh` and `tpm-crypto-mount.sh` are written in Bash. Detailed instructions how to use it, can be found in the provided programming-documentation.txt file.

Conclusion

This bachelor thesis consists of four chapters. In the first chapter is described the architecture and the main functions of the TPM and TEE as a trusted measurement, reporting and storage and all the components needed to handle all of these requirements. The aim of this chapter was to show the credibility of these functions and what this credibility is based on. In a TPM section 1.2 I have mainly described the PCR registers, because the whole practical part is built on their use. The overall effort in this chapter was to create a theoretical basis for the reader about the TPM and TEE and its provided functions.

The second chapter describes all the tools used for commissioning and communication with TPM (TrouSerS, tpm-tools, TrustedGRUB, cryptsetup). I was mainly concerned with cryptsetup, due to block device encryption used in my practical part.

The primary goal of this thesis was a specific example of a TPM chip in the form of a simple program, because more complex programs and use was seemed to be a bit too much for introducing the reader to this issue. This program was extensively analyzed and its description is presented in chapter 3. The solution of encryption was found to be sufficiently secure and practical for use. So I suggested a solution that is practical for everyone, and I will like my scripts, that are introduced in chapter 4, to be used by others as well as I will be using it to decrypt my USB disk.

I would like to indicate the enhanced usage of this script and an idea to continue: write a script that handles system update for Linux Mint and Ubuntu.

Thanks to this work, I successfully wrote scripts, which tested the behavior of TPM with the TrouSerS environment and I learned to encrypt and decrypt the disk.

Bibliography

[1] Sadeghi, A. R. Introduction to Trusted Computing. 2016, [accessed 2017-07-18]. Available from: https://www.trust.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_TRUST/LectureSlides/STC-WS2016/02-Introduction_to_Trusted_Computing.pdf

[2] Schmidt, M. Trusted computing a Linux. [accessed 2017-07-18]. Available from: http://docplayer.cz/46636277-Trusted-computing-a-linux-michal-schmidt-red-hat.html

[3] Trusted Platform Modules revisited. [Online]http://deveck.net/node/30, [accessed 2017-07-18].

[4] Dinh, T. T. A.; Ryan, M. D. Protected storage and Root of Trust for Storage (RTS). [accessed 2017-07-18]. Available from: https://www.cs.bham.ac.uk/~mdr/teaching/modules/security/lectures/TrustedComputingTCG.html

[5] McGillion, B. *Open virtual trusted execution environment*. Master's thesis, Tampere University of Technology, March 2016, [accessed 2017-07-18]. Available from: https://dspace.cc.tut.fi/dpub/bitstream/handle/123456789/23823/Dettenborn.pdf

[6] ARM Limited. *ARM Security Technology Building a Secure System using TrustZone® Technology*. 2005-2009, [accessed 2017-07-18]. Available from: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c

[7] Fruhwirth, C. *LUKS On-Disk Format Specification*. December 2008, [accessed 2017-07-18]. Available from: http://tomb.dyne.org/Luks_on_disk_format.pdf

[8] Choinyambuu, S. *A Root of Trust for Measurement.* Mse project report, Hochschule Rapperswil, June 2011, [accessed 2017-07-18]. Available from: `http://security.hsr.ch/mse/projects/2011_Root_of_Trust_for_Measurement.pdf`

[9] Trusted Computing Group. *Design principles.* March 2011, [accessed 2017-07-18]. Available from: `https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-1-Design-Principles_v1.2_rev116_01032011.pdf`

[10] Arthur, W.; Chalenner, D.; et al. *A Practical Guide to TPM 2.0.* Apres Open, 2015, ISBN 978-1430265832.

[11] GNU Free Documentation License 1.3 or later. *Disk encryption.* [accessed 2017-07-18]. Available from: `https://wiki.archlinux.org/index.php/disk_encryption`

[12] Ryan, M. *Introduction to the TPM 1.2.* diploma thesis, University of Birmingham, March 2009, [accessed 2017-07-18]. Available from: `ftp://ftp.cs.bham.ac.uk/pub/authors/M.D.Ryan/08-intro-TPM.pdf`

[13] Vasudevan, A.; Owusu, E.; et al. *Trustworthy Execution on Mobile Devices: What security properties can my mobile platform give me?* Springer, Berlin, Heidelberg, 2012, ISBN 978-3-642-30921-2.

[14] Martin, A. Trusted Infrastructure 101. 2011, [accessed 2017-07-19]. Available from: `https://www.cylab.cmu.edu/tiw/slides/martin-tiw101.pdf`

[15] Martin, A. The ten-page introduction to Trusted Computing. November 2008, [accessed 2017-07-18]. Available from: `https://www.cs.ox.ac.uk/files/1873/RR-08-11.PDF`

[16] Trusted Computing Group. *TCG Glossary.* 2012, [accessed 2016-12-4]. Available from: `http://www.trustedcomputinggroup.org/wp-content/uploads/TCG_Glossary_Board-Approved_12.13.2012.pdf`

[17] Smith, S. W. *Trusted Computing Platform.* Springer, ISBN 978-7-302-13174-8, 2014.

[18] Schellekens, D. *Design and Analysis of Trusted Computing Platforms.* Dissertation thesis, Katholieke Universiteit Leuven – Faculty of Engineering Science, December 2012, [accessed 2017-10-21]. Available from: `https://www.esat.kuleuven.be/cosic/publications/thesis-219.pdf`

[19] Trusted Computing Group. [accessed 2017-10-21]. Available from: `https://trustedcomputinggroup.org/tpm-main-specification/`

[20] Trusted Computing Group. [accessed 2017-10-21]. Available from: `https://trustedcomputinggroup.org/tpm-library-specification/`

[21] Goldman, K.; Potter, S. *SHA-1 Uses in TPM v1.2*. Trusted Computing Group, April 2010, [accessed 2017-12-08]. Available from: `https://www.trustedcomputinggroup.org/wp-content/uploads/SHA1-Impact_V2.0.pdf`

[22] SBS Implementers Forum. *System Management Bus (SMBus) Specification*. August 2000, [accessed 2017-07-18]. Available from: `http://tomb.dyne.org/Luks_on_disk_format.pdf`

[23] Stetsko, A. *Principy a využití modulu důvěryhodné platformy*. Master's thesis, Masarykova univerzita, 2007, [accessed 2017-10-27]. Available from: `https://is.muni.cz/th/184905/fi_m/thesis.pdf`

[24] Trusted Computing Group. *TCG PC Client Specific Implementation Specification For Conventional BIOS*. July 2005, [accessed 2017-07-18]. Available from: `https://trustedcomputinggroup.org/wp-content/uploads/PC-Client-Implementation-for-BIOS.pdf`

[25] Ge, H. A Method to Implement Direct Anonymous Attestation. [accessed 2017-07-18]. Available from: `https://eprint.iacr.org/2006/023.pdf`

[26] Information Technologies for IPR Protection. `http://www.cmlab.csie.ntu.edu.tw/~ipr/ipr2006/data/lecture/Lecture13%20-%20Trusted%20Platform%20Module%20(TPM).pdf`, [accessed 2017-07-18].

[27] Device Specifications TEE system Architecture. [accessed 2017-10-27]. Available from: `https://www.globalplatform.org/specificationsdevice.asp`

[28] Coojimans, T. *Secure Key Storage and Secure Computation in Android*. Master's thesis, Radboud University Nijmegen, June 2014, [accessed 2017-07-18]. Available from: `www.ru.nl/publish/pages/769526/scriptie_tim_cooijmans.pdf`

[29] GlobalPlatform made simple guide: Trusted Execution Environment (TEE) Guide. [accessed 2017-10-27]. Available from: `https://www.globalplatform.org/mediaguidetee.asp`

[30] ARM Processors. [accessed 2017-07-18]. Available from: `http://www.arm.com/products/processors`

[31] González, J. *Operating System Support for Run-Time Security with a Trusted Execution Environment.* Dissertation thesis, IT University of Copenhagen, January 2015, [accessed 2017-07-18]. Available from: `https://en.itu.dk/~/media/en/research/phd-programme/phd-defences/2015/javiergonzalez_phdprintversion-pdf`

[32] Open-TEE. [Online] `https://open-tee.github.io/`, Visited 31.12.2017.

[33] Qualified Products. [accessed 2017-07-18]. Available from: `https://globalplatform.org/complianceproducts.asp`

[34] Trusted Computing Group. [accessed 2017-07-18]. Available from: `https://trustedcomputinggroup.org/work-groups/mobile/`

[35] TRUECRYPT Free open - source on-the-fly encryption USER'S GUIDE The TrueCrypt User's Guide for v7.1a. February 2012, [accessed 2017-07-18]. Available from: `https://www.grc.com/misc/truecrypt/TrueCrypt.htm`

[36] GNU Free Documentation License 1.3 or later. *dm-crypt/Device encryption.* [accessed 2017-12-25]. Available from: `https://wiki.archlinux.org/index.php/Dm-crypt/Device_encryption`

[37] Daemen, V., J.; Rijmen. *The Design of Rijndael.* NJ, USA: Springer-Verlag New York, 2002, ISBN 3540425802.

[38] Hoang, V. T.; Rogaway, P. *On Generalized Feistel Networks. Advances in Cryptology.* Springer, Berlin, Heidelberg, 2010, ISBN 978-3-642-14623-7.

[39] Schneier, B.; Kelsey, J.; et al. Twofish: A 128-Bit Block Cipher. June 1998, [accessed 2017-07-18]. Available from: `https://www.schneier.com/academic/paperfiles/paper-twofish-paper.pdf`

[40] Anderson, R.; Biham, E.; et al. Serpent: A Proposal for the Advanced Encryption Standard. Technical report, Cambridge University, University of Bergen, Technion, September 1998.

[41] Block Cipher Modes of Operation. [accessed 2017-07-18]. Available from: `https://www.tutorialspoint.com/cryptography/block_cipher_modes_of_operation.htm`

[42] Voloshynovskiy, S.; Pereira, S.; et al. Watermark attacks. 1999, [accessed 2017-07-18]. Available from: `http://cvml.unige.ch/publications/postscript/99/VoloshynovskiyPereiraPun_eww99.pdf`

[43] IEEE, New York, New York 10016-5997, USA. *IEEE P1619™/D16 Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices.* 2007. Available from: `http://grouper.ieee.org/groups/1619/email/pdf00086.pdf`

[44] trousers. [accessed 2017-07-18]. Available from: https://sourceforge.net/projects/trousers/files/trousers

[45] Trusted Computing Group. *TCG PC Client Specific TPM Interface Specification (TIS).* July 2005, [accessed 2017-07-18]. Available from: https://trustedcomputinggroup.org/wp-content/uploads/TCG_PCClientTPMSpecification_1-20_1-00_FINAL.pdf

[46] Yoder, K. *tcsd.* TCG Software Stack, [accessed 2017-07-18]. Available from: https://linux.die.net/man/8/tcsd

[47] tpm-tools. 2006, [accessed 2017-07-18]. Available from: https://sourceforge.net/p/trousers/tpm-tools/ci/master/tree

[48] trousers-users. *tpm_nvdefine.* [accessed 2017-07-18]. Available from: http://manpages.ubuntu.com/manpages/xenial/man8/tpm_nvdefine.8.html

[49] Segall, A. Using the TPM: Data Protection and Storage. [accessed 2017-07-18]. Available from: http://opensecuritytraining.info/IntroToTrustedComputing_files/Day2-2-data-storage.pdf

[50] Ubuntu manuals. [accessed 2018-01-03]. Available from: http://manpages.ubuntu.com/manpages/wily/man8/

[51] Mitchell, C. *Trusted Computing.* IET, 2008, ISBN 978-0-86341-525-8.

[52] Corbin, K. LUKS with TPM in Ubuntu. [accessed 2017-07-18]. Available from: http://tomkowapp.com/2016/04/09/Ubuntu-TPM-encryption/

[53] Rohde, S. TrustedGRUB2. [accessed 2017-07-18]. Available from: https://github.com/Rohde-Schwarz-Cybersecurity/TrustedGRUB2

[54] Drake, D. *Writing udev rules.* 2008, [accessed 2017-07-18]. Available from: http://www.reactivated.net/writing_udev_rules.html

[55] GNU Free Documentation License 1.3 or later. *udev ArchLinux.* [accessed 2017-07-18]. Available from: https://wiki.archlinux.org/index.php/udev

[56] Yoder, K.; Goldman, K.; et al. tpm-luks. [accessed 2017-07-18]. Available from: https://github.com/shpedoikal/tpm-luks/blob/master/tpm-luks/tpm-luks-update

Appendix A

Acronyms

AES Advanced Encryption Standard

AIK Attestation Identity Keys

API Application Programming Interface

BIOS Basic Input/Output System

CA Certification Authority

CPU Central Processing Unit

CRTM Core Root of Trust for Measurements

DAA Direct Anonymous Attestation

DES Data Encryption Standard

DRTM Dynamic Root of Trust for Measurements

DSA Digital Signature Algorithm

ESCD Extended System Configuration Data

EK Endorsement Key

EXT2 Second Extended Filesystem

GNU GNU's Not Unix!

GNU GPL GNU General Public License

HTTP Hypertext Transfer Protocol

HW Hardware

IEC International Electrotechnical Commission

IEEE Institute of Electrical and Electronics Engineers

iOS iPhone OS

IPL Initial Program Load

ISO International Organization for Standardization

IV Initialization Vector

LBA Logical Block Addressing

LPC Low Pin Count

LUKS Linux Unified Key Setup

MBR Master Boot Record

NIST National Institute of Standards and Technology

NVRAM Non-volatile access memory

OS Operating System

PC Personal Computer

PCR Platform Configuration Registers

POST Power-On Self-Test

PBKDF2 Password-Based Key Derivation Function 2

PRNG Pseudorandom Number Generator

RAM Random Access Memory

REE Rich Execution Environment

ROM Read-Only Memory

RNG Random Number Generator

RSA Rivest, Shamir, Adleman cipher

RTM Root of Trust for Measurements

RTR Root of Trust for Reporting

RTS Root of Trust for Storage

SCSI Small Computer System Interface

SHA Secure Hash Algorithms

SMBus	System Management Bus
SML	Stored Measurement Log
SoC	System on Chip
SRK	Storage Root Key
SRTM	Static Root of Trust for Measurements
SVM	Secure Virtual Machine
SW	Software
TC	Trusted Computing
TCPA	Trusted Computing Platform Alliance
TDDL	TCG Device Driver Library
TCG	Trusted Computing Group
TCS	TCG Core Services
TEE	Trusted Execution Environment
TPM	Trusted Platform Module
TSP	TCG Service Provider (TSP)
TSS	Trusted Software Stack
TTP	Trusted Third Party
TXT	Trusted Execution Technology
USB	Universal Serial Bus
UUID	Universally Unique Identifier
VFAT	Virtual File Allocation Table
VM	Virtual Machine
XEX	Xor-Encrypt-Xor
XML	Extensible Markup Language

Appendix B

Content of enclosed CD

```
readme.txt ........................ the file with CD contents description
src ........................... directory of source codes written in Bash
  scripts .......................... implementation sources of scripts
  thesis .............. the directory of LaTeX source codes of the thesis
text ......................................... the thesis text directory
  BP_Holoubkova_Andrea_2018.pdf ...... the thesis text in PDF format
```