POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Secure Boot and Monitoring for Embedded System

Supervisor
prof. Antonio Lioy

Damiano Zappulla

Academic Year 2021-2022

To my grandparents, who watch over me

Summary

Remote attestation is the activity of making a claim about properties of a target by supplying evidence to an appraiser over a network [1]. Thanks to Linux Integrity Measurement Architecture (IMA) it is possible to maintain the chain of trust measurement up to the application layer. The goals of the kernel integrity subsystem are to detect if files have been accidentally or maliciously altered, both remotely and locally, appraise a file's measurement against a "good" value stored as an extended attribute, and enforce local file integrity [2]. This thesis firstly describes history, architecture, version and capabilities of Trusted Platform Module (TPM), crucial component to perform remote attestation and, inside the second part, Keylime, an open-source tool for bootstrapping and maintaining trust in the cloud, is presented and evaluated. The practical part of this paper covers the installation, configuration and evaluation of TPM tools and Keylime, the activation and testing of Linux IMA with Keylime itself.

Acknowledgements

I would like to express my gratitude to Professor Antonio Lioy for offering an interesting topic. I would also manifest my gratitude to my family and my friends for their support during my entire studies.

Contents

Chapter 1

Introduction

1.1 Motivation

With the spread of Internet infrastructure, the lack of a trustworthy infrastructure has been an important obstacle for the development of modern platform. The increase in software complexity inevitably leads to an increase in vulnerabilities within the code. In this context, the construction of a platform whose status can be monitored becomes of fundamental importance. Following the guidelines proposed by the Trusted Computing Group (TCG) is critical to achieving this goal.

Trusted computing architecture depends on Trust Platform Module (TPM). TPM is the base of the trust chain and the trusted root throughout the trusted boot process, which records and transfers trusted states in end system [3].

Nowadays, just running the trusted boot process is not enough, it is necessary to extend the verification to the application level as well. This objective is fulfilled by the Linux Integrity Measurement Architecture (IMA), that was introduced in Linux 2.6.30, as part of an overall Linux Integrity Subsystem. With IMA is possible to maintain the chain of trust measurements up to the application layer [2]. Obviously, extending the verification process to the application level is not at all simple because, unlike the bootstrap process, an operating system handles a large variety of executable content (kernel, kernel modules, binaries. shared libraries, scripts, plugins). With Linux IMA, all executable content that is loaded onto the Linux system is measured before execution and these measurements are protected by the Trusted Platform Module (TPM).

1.2 Objectives

The main objective of the thesis is to highlight the need to perform remote attestation on particularly sensitive machines and to provide a trustworthy and reliable solution to to perform remote attestation integrity verification with Linux IMA enabled and test it.

This proposed solutions focus on providing remote attestation integrity verification during runtime. Therefore providing the possibility for the Keylime Verifier to perform attestation at any time and with a multiple number of times without interrupting the attested software's execution. In addition to this the Verifier will be able to manage and attest an arbitrary number of remote devices at the same time.

1.3 Structure

This thesis is divided into four chapters:

- Background and Technologies (chapter 2) will introduce all theoretical concepts necessary for understanding the work done.
- Linux IMA (chapter 3) describes the functionality of the so called kernel module.
- Keylime (chapter 4) introduces the software used to perform the remote attestation process, describes its architecture, framework and configuration on the platform used to perform the tests.
- Testing (chapter 5) contains the explanation of the tests carried out to evaluate the attestation software.

Chapter 2

Background and Technologies

Trusted Computing is a general term describing hardware assisted security technologies used to secure software. Security technologies are necessary because nowadays systems have a growing number of components from multiple sources and not all the sources can be verified or trusted equally.

In general, security of a system depends on a set of hardware and software components which is called the Trusted Computing Base (TCB) [4]. A vulnerability in some part of TCB would compromise the security of whole system. On the other hand, misbehaving hardware or vulnerability in software outside of TCB must not affect the security of the whole system.

Hardware based solutions to trusted computing vary greatly as they have different objectives and constraints. A Trusted Platform Module (TPM) is a secure crypto-processor, which is used as key storage, to authenticate the platform, and to ensure platform integrity.

2.1 History of the TPM

In 1981 the Department of Defense founded the Computer Security Initiative (CSI) which released a paper in which are defined Trusted Computing Systems: systems that "employ sufficient hardware and software integrity measures to allow its use in processing multiple levels of classified or sensitive information" [5].

In 1985, inside a paper called "Trusted Computer System Evaluation Criteria" [4] we can find the first definition of the Trusted Computing Base (TCB) of a computer system: collection of system resources (hardware and software) that is responsible for maintaining the security policy of the system.

In 1999 the Trusted Computing Platform Alliance (TCPA), a group of various technology companies including Compaq, Hewlett-Packard, IBM, Intel and Microsoft, was formed to develop trust and security in the computing platforms. The group's aim is to provide the industry with a clear direction that facilitates trust in computing platforms and environments. TCPA's original goal is to develop a Trusted Platform Module (TPM), an hardware module whose aim is to enable trusted computing features [6].

Trusted Computing defines schemes for establishing trust in a platform that are based on identifying its hardware and software components. The Trusted Platform Module (TPM) provides methods for collecting and reporting these identities [7].

In 2002 were published TCPA Main Specification Version 1.1b [8] in which is described a minimal TPM intended to be physically affixed to the motherboard of a PC, with a restricted command set which included only the main security functions.

In 2003 the TCPA became the Trusted Computing Group (with fourteen members) and published in 2004 TPM 1.2 Main Specification with the goal to overcome the problems of version 1.1b [9]. With TPM specification 1.2 manufacturers such as IBM, HP and DELL began to integrate this technology in on most x86-based client PC and servers.

Because of the first attack on the SHA-1 digest algorithm, widely used in TPM 1.2, TCG started to work on a new version of TPM and in 2015 TCG released TPM 2.0 Library Specification [10]. The main feature of TPM 2.0 is algorithms agility: TCG designers chose to not define any algorithm in the TPM 2.0, but rather to incorporate an algorithm identifier that allowed to use a wide range of cryptographic algorithms and to enlarge the set of algorithms over time without the need to change the specification.

2.2 TPM: Trusted Platform Module

A Trusted Platform Module, or TPM, is a passive (needs to be driven by the CPU) secure crypto-processor that is designed to carry out cryptographic operations. A TPM must be tamper-resistant (not tamper-proof) and be certified Common Criteria EAL4+ [11]. A TPM is a physical or embedded security technology (microcontroller) that resides on a computer's motherboard or in its processor. The TPM enables a series of cryptographic functionalities like:

- Keys management
- Hash computation
- Encryption/decryption functionalities
- Sign/verify functionalities
- Hardware number generator
- Binding operation: TPM offers protection of the message by means of asymmetric cryptography using encryption keys generated and maintained by the TPM. Thus, a message encrypted using a particular TPM's public key is decryptable only by using the private key of the same TPM.
- Sealing operation: Similar to binding, but in addition, the encrypted messages produced through binding are only decryptable in a certain platform state (defined through PCR values) to which the message is sealed. This ensures that an encrypted message can only be decrypted by a platform which is in a certain prescribed state.

The TPM was designed by the Trusted Computing Group and standardized by the International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) in 2009 as ISO/IEC 11889. TPM specification [12] was aimed at addressing the following major issues in the industry:

- Identification of devices: Without TPM devices were identified through MAC addresses or IP addresses. TPM characteristics, like the Endorsement Key, allow a device to be uniquely identified.

- Secure generation of keys: TPM keys generation relies on an hardware random number generator.
- Secure storage of keys: Keeping good keys secure, particularly from software attacks, is a big advantage that the TPM design brings to a device.
- Device health attestation: TPM is the best way to ensure system attestation and integrity.

2.2.1 Architecture

Functionalities of TPM may be understood better analyzing TPM architecture. The main components [10] of the hardware TPM chip and description of each components are shown in the below figure 2.1.

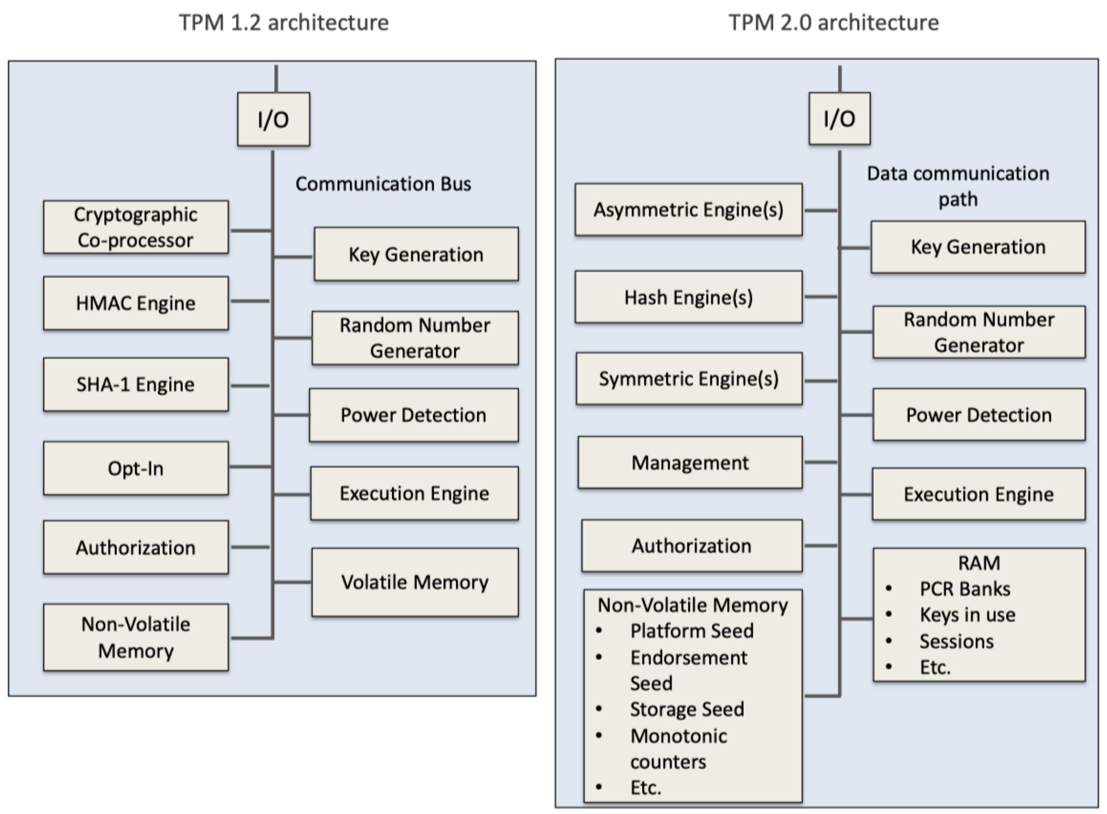


Figure 2.1. TPM 1.2 and 2.0 architectures [10]

I/O Block

I/O Block is a communication bus for managing the data flow and communication between the components. This is not a general I/O block and it is not accessible by the operating system. TPM plays the role as a slave device through this interface, which means it can respond only to the received commands from TPM Device Driver.

Execution Engine

Execution Engine (usually CPU) is responsible to run commands from I/O and program code which is in the TPM. The program code that resides on the TPM is the actual root of trust for integrity measurements.

Power Detection

Power Detection is in charge of handle TPM power states.

Authorization Subsystem

When a TPM 2.0 command tries to access to a shielded location, Authorization Subsystem is in charge of performing authorization checks before allowing the execution of a command.

Volatile Storage

Volatile Storage is a temporary memory which depends on the power of the device and will lose its content upon reboot of the TPM. This memory contains data related to temporary state, authorization sessions and entities such as keys and data objects loaded in the TPM from external memory and PCR Banks.

Non-volatile Storage

Non-volatile Storage contains Shielded Locations which can be accessed only through protected capabilities. Having NVRAM provides the following:

- Storage for root keys for certificate chains
- Storage for an endorsement key (EK)
- Storage for the state of the machine
- Storage for decryption keys used before the hard disk availability (e.g. key used for a self-encrypting drive)

Random number generator

Random number generator is a device that generates random numbers from a physical process, rather than by means of an algorithm. Such devices are often based on microscopic phenomena that generate low-level, statistically random "noise" signals, such as thermal noise. According to the specification, the TPM can be a good source of unpredictable random numbers even without having to require a genuine source of hardware entropy.

Key Generator (KG)

Key Generator is based on TPM's own Random Number Generator and doesn't rely on any external sources of randomness. Therefore, it alleviates the weaknesses of key security based on software with an insufficient source of entropy or software with weak random number generators. In TPM 2.0, keys can be generated from a seed, using TPM's RNG or imported from another TPM. Generating TPM keys is the most time-consuming cryptographic calculation in most of the cases.

Asymmetric and Symmetric Engines

Engines used for all the operations that involve asymmetric and symmetric key like signing, verifying, encrypting and decrypting.

Hash engine

Hash engine performs hashing operations and could be used both directly by the system and as part of other TPM operations (like as part of the key derivation function)

Primary Seed

A Primary Seed is a large, random value that is persistently stored in a TPM; it is never stored off the TPM in any form. Primary Seeds are used in the generation of symmetric keys, asymmetric keys, other seeds, and proof values [10]. Such keys can be derived from primary objects, which in turn are created from primary seed by using Key Derivation Function (KDF). There are three persistent primary seeds for platform, endorsement and storage hierarchies, as well as, one non-persistent primary seed for the Null hierarchy. The seeds are embedded inside the TPM during manufacturing [7].

Platform Configuration Registers

TPM is able to report current state of the system; in order to store this integrity metrics and for attestation purposes, a TPM contains a special set of registers called PCR (Platform Configuration Register). Storing a value in a PCR is called an extend operation, which combines the previous hash value and new measurement with a one-way hash function.

A TPM contains 24 PCRs, where measurements from different platform components are stored in distinct registers. The platform can be shown to be in some verified state by verifying the value of each register.

Only two operations are allowed on PCRs: register extending and register reset. Reset operation simply set the target PCR register to zero. Extend operation (defined in Equation 2.1) concatenates the existing PCR value with a new measurement, securely hashes that value, and stores the resulting hash in the register.

$$PCR_{new} := HASH_{alg}(PCR_{old}||digest) \tag{2.1}$$

Values of PCR can be described as an unbalanced binary hash tree, where new value is added as a new root node and left child is the old value and the right child is the input parameters [13]. Figure 2.2 contains a visualization of PCR values interpreted as binary hash tree where the hash function is SHA-256.

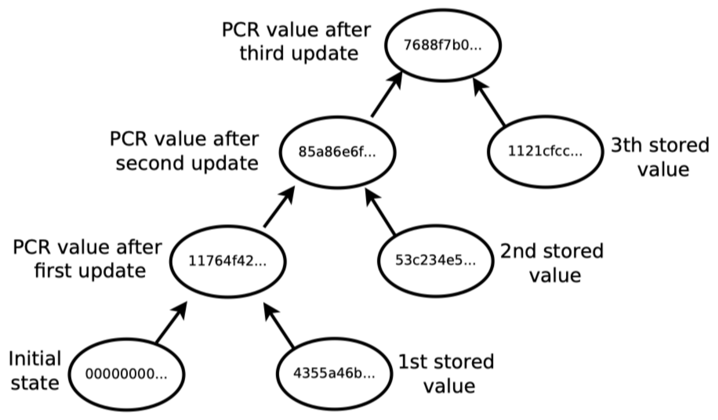


Figure 2.2. PCR interpreted as binary hash tree [14]

Endorsement Key (EK)

An Endorsement Key is a special purpose TPM-resident RSA key that is never visible outside of the TPM. Because the EK can only be used for encryption, possession of the private EK can only be proved indirectly, by using it to decrypt a value that has been encrypted with the public EK.

This key is a fundamental component of the TPM and it is derived from the Endorsement seed that is embedded under the Endorsement hierarchy in the tamper resistant non-volatile memory by TPM manufacturer.

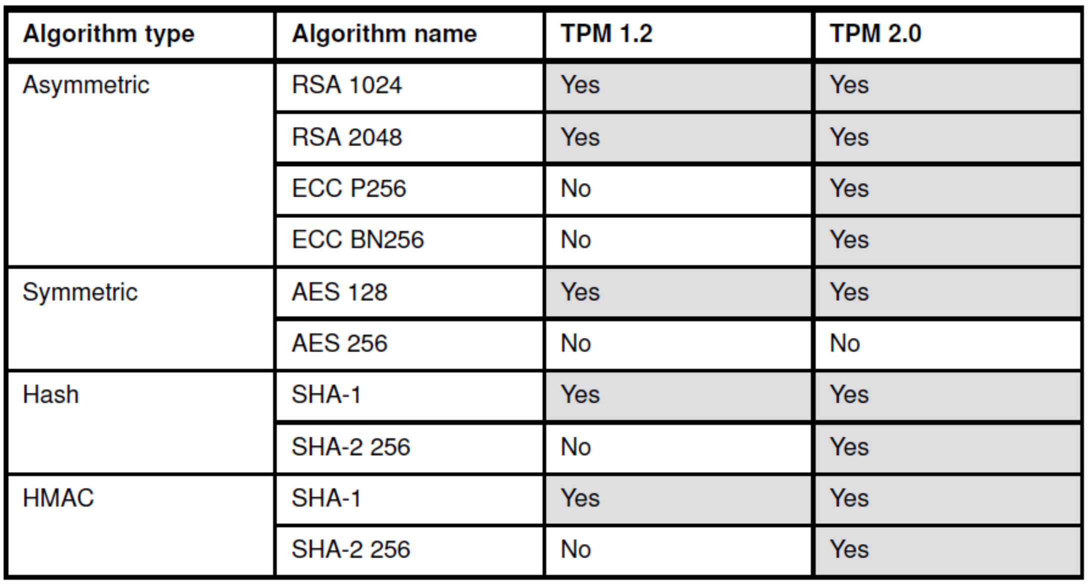
Attestation Identity Key (AIK)

An Attestation Identity Key is a special purpose TPM-resident RSA key that is used to provide platform authentication based on the attestation capability of the TPM. AIK is a 2048 bit key size, generated under the Endorsement hierarchy and signed by the EK of the TPM. The AIK is designed to sign data that is generated exclusively by the TPM. The main purpose of the TPM is to be used in the attestation of TPM which is a part of the secure boot process. It can also be used in authentication of a platform that hosts a TPM to a remote server.

2.2.2 TPM Versions

There are two main versions of TPM specifications: TPM 1.2 Main Specification [9] and TPM 2.0 Library Specification [10]. TPM 1.2 has a one-size-fits-all specification, while the 2.0 version has platform-specific specifications that define which parts of the library are mandatory or optional [15]. The TPM 2.0 implementations enable the same features as 1.2, plus several more [12]:

- Algorithm agility
- Enhanced authorization
- Quick key loading



Algorithm type	Algorithm name	TPM 1.2	TPM 2.0
Asymmetric	RSA 1024	Yes	Yes
	RSA 2048	Yes	Yes
	ECC P256	No	Yes
	ECC BN256	No	Yes
Symmetric	AES 128	Yes	Yes
	AES 256	No	No
Hash	SHA-1	Yes	Yes
	SHA-2 256	No	Yes
HMAC	SHA-1	Yes	Yes
	SHA-2 256	No	Yes

Figure 2.3. TPM 1.2 - 2.0

- Non-brittle PCRs
- Flexible management
- Identifying resources by name

Algorithms agility

As for algorithms on TPM 1.2, SHA-1 and RSA are required, while the AES is optional. With TPM 2.0, SHA-1 and SHA-256 are required for hashes. RSA and ECC with Barreto-Naehrig 256 bit curve and a NIST P-256 curve are used for public-key cryptography and asymmetric digital signature generation and verification in TPM 2.0 [15]. As for symmetric digital signature generation, the TPM 2.0 is using the HMAC, and 128 bit AES for symmetric-key algorithms [16].

TPM 2.0 introduce a feature called Algorithm Agility that allows to implement a set of algorithms compatible with specific use cases. This feature also makes it much easier to update supported algorithms without modifying the specification.

Enhanced authorization

Authentication mechanism has been radically changed between the two versions of the TPM. In TPM 1.2 it was intricate: keys had two authorizations: one for use of the key and one to make duplicates of the key (called migration in the TPM 1.2 specification). Additionally, keys could be locked to localities and values stored in PCRs. Similarly, the NVRAM in TPM 1.2 could be locked to PCRs and particular localities, and to two different authorizations-one for reading and one for writing [12].

TPM 2.0 authorization mechanism, called Enhanced Authorization uses the following kinds of authorizations:

- Password (in the clear)

- HMAC key (also present in TPM 1.2)
- Signature (e.g. smart card)
- Signature with additional data (e.g. fingerprint reader)

EA system can grant authorization in case of verification of the following conditions:

- PCR values as a proxy for the state of the system
- Locality as a proxy for where a particular command came from
- Time
- Internal counter values
- Value in an NV index
- NV index
- Physical presence

Quick key loading

With TPM 1.2, the first loading of a key required a time-consuming private-key decryption by using another private key, the "parent" key, used to cipher user's key. In order to speed up next loading, it was possible to save to cache loaded keys ciphered with a symmetric algorithm. Once the TPM was turned off, however, the symmetric key was erased, thus the next loading of the user's key needed slow asymmetric decryption.

TPM 2.0 specification allows to use of symmetric encryption for directly protecting user data. There is little reason to cache keys out to disk because loading them is usually as fast as recovering them from a cached file. Thanks to the speeding up of the loading of the keys it is possible for multiple users to use a TPM without noticing a long delay.

Non-brittle PCRs

From TPM 1.2 it is possible to perform sealing operation. But if keys or data are locked to a PCR that represents the BIOS of a system, it's tricky to upgrade the BIOS: this is known as *PCR fragility*.

In the TPM 2.0 specification, you can seal things to a PCR value approved by a particular signer, for example the OEM of the system software, instead of to a particular PCR value. It is therefore possible to decrypt a message only if PCRs are in a state approved (via a digital signature) by a particular authority.

Flexible management

TPM 1.0 allowed only two types of authentications: owner authorization and the storage root key (SRK) authorization. SRK authorization however was usually the well-known secret (20 B of 0s) and therefore the owner authorization was used for many purposes, thus it was very difficult to manage different roles independently.

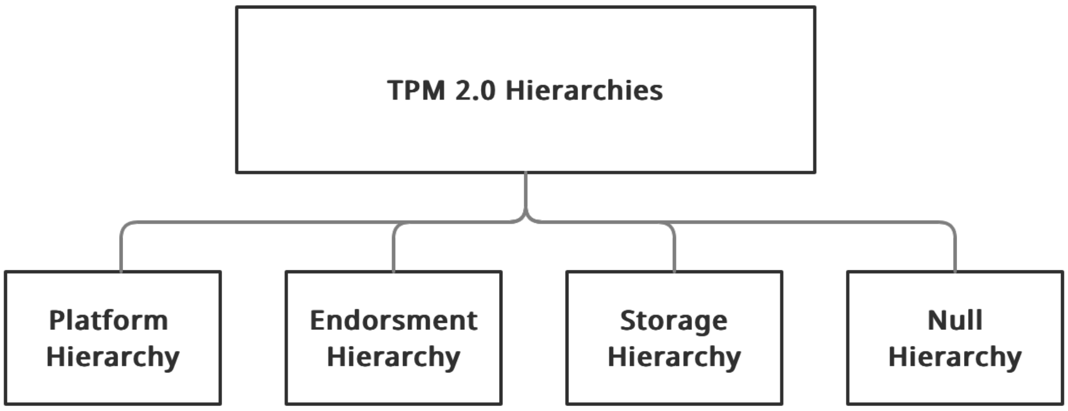


Figure 2.4. TPM hierarchies

TPM 1.2 allowed to delegate the owner-authorization role to different entities using the *Delegate* commands, but those commands were fairly complicated and used up valuable NVRAM space [12].

In TPM 2.0, the roles represented by the various uses are separated in the specification itself. As shown by figure 2.4, TPM 2.0 has four hierarchies: Platform, Storage, Endorsement and Null hierarchy. Each of the hierarchies has a primary seed which is embedded in the TPM by the TPM manufacturer.

- Platform Hierarchy: under the control of platform manufacturer, represented by the early boot code inserted in the platform by the manufacturer (BIOS).
- Endorsement Hierarchy: under the control of a privacy administrator, who might be the end user. TPM and administrator certify that primary keys in this hierarchy are constrained to an authentic TPM attached to an authentic platform [12].
- Storage Hierarchy: Replicates the TPM 1.0 family SRK for the most part
- Null Hierarchy: like other hierarchies, but it cannot be disabled. The seed of null hierarchy is not persistent and upon each TPM reboot, a new seed with different value is generated. So new primary objects can be created from this seed, the authorization is a password of length equal to zero and the policy is empty.

Identifying resources by name

With TPM 1.2, resources were identified by handles instead of by a cryptographically bound name. Thus with two resources with the same authorization policy, low level software could be hacked in order to change the handle identifying the resource and to authorize a different action.

Since TPM 2.0, resources are identified by names cryptographically bound to them. It is possible to use TPM key to sign the name in order to provide an evidence that the name is correct [12].

Trust Element	Security Level	Security Features	Cost	Application
Discrete TPM	Highest	Tamper Resistant HW	$$$	Critical Systems
Integrated TPM	Higher	Hardware	$$	Gateways
Firmware TPM	High	Tee	$	Entertainment
Software TPM	NA	NA	¢	Testing
Virtual TPM	High	Hypervisor	¢	Cloud

Table 2.1. TPM different implementations [17]

2.2.3 TPM Implementation

With TPM 2.0, TCG created a library specification that describes all the commands/features that could be implemented and might be needed in platforms from servers to laptops to embedded systems [6]. Each platform can choose the features needed and the level of security or assurance required. In this way, TPM 2.0 is much more flexible than the original TPM specification [17]. Table 2.1 summarizes all implementation of TPM 2.0.

Discrete TPM

Discrete TPM provides the highest level of security, as might be needed for a TPM used to secure the brake controller in a car. The intent of this level is to ensure that the device it is protecting does not get hacked via even sophisticated methods. To accomplish this, a discrete chip is designed, built and evaluated for the highest level of security that can resist tampering with the chip, including probing it and freezing it with all sorts of sophisticated attacks.

Integrated TPM

Integrated TPM is an hardware TPM but integrated into a chip that provides functions other than security. The hardware implementation makes it resistant to software bugs, however, this level is not designed to be tamper-resistant.

Firmware TPM

TPM implemented in protected software. The code runs on the main CPU, so a separate chip is not required. While running like any other program, the code is in a protected execution environment called a trusted execution environment (TEE) that is separated from the rest of the programs that are running on the CPU. By doing this, secrets like private keys that might be needed by the TPM but should not be accessed by others can be kept in the TEE creating a more difficult path for hackers.

Virtual TPM

Virtual TPM is a software solutions provided by a hypervisor. Since they are executed in an environment that is hidden from the software running inside, they provide the same level of security as firmware TPMs.

Software TPM

Software TPM is a TPM implemented as a software emulator. Software TPM is open to many vulnerabilities, not only tampering but also the bugs in any operating system running it and should only be used for testing.

2.3 Remote Attestation

A trusted component or a trusted platform is one that behaves as expected. This does not mean that it is absolutely good or secure, it only means that the component behaves as programmed. Trust is not the same as good/secure, it regards real behavior against the expected one. In order to achieve that we perform attestation, a verifiable evidence of the platform's state.

2.3.1 Roots of Trust

Root of Trust is defined as a source that can always be trusted within a cryptographic system because its misbehavior cannot be detected. A TPM can work only if the RoTs are properly implemented.

A Trusted Platform, according to Trusted Computing Group specifications, must provide three different RoT:

- Root of Trust for Storage (RTS)
- Root of Trust for Measurement (RTM)
- Root of Trust for Reporting (RTR)

Root of Trust for Storage (RTS)

Root of Trust for Storage is a special portion of memory which is shielded, no other entities except TPM can modify the value inside it.

Root of Trust for Measurement (RTM)

Root of Trust for Measurement measures and sends integrity measurement to RTS. The TPM cannot implement the RTM, since it is conceived as a slave device that receives commands. RTM of a platform is the Core Root of Trust for Measurement (CRTM) which is executed usually by the CPU.

Root of Trust for Reporting (RTR)

Root of Trust for Reporting s responsible for reliably reporting values stored in the RTS. Typically an RTR report is a digitally signed digest calculated on the values of some Shielded Locations within a TPM.

Chain of Trust

Integrity should be maintained in any running software on networking device to provide the security of the system. This can be obtained through Chain of Trust model. Each step of model since start of the algorithm, check next steps before it executes, and can process as many steps as required. The first step of the model is executed by Root of Trust which starts the chain of the continuous checking.

2.3.2 Trusted Platform Boot

Trusted boot is one of the core functions of trusted computing platform. During system boot process, with the support of a TPM, it is possible to build a trusted running environment with the verification of the integrity of the whole hardware and software [3]. Trusted boot is based on three key points:

1. The chain of trust must be established sequentially. The executable entity can be loaded only after its integrity is verified by trusted computing base.
2. All the metrics and calls involved in the process of the establishment of the trust chain will eventually be completed by TPM.
3. During the establishment of chain of trust, TPM is responsible for ensuring the integrity and confidentiality of data, thus all important data must be stored and sealed inside TPM.

Figure 2.5 shows the Linux trusted boot process based on TPM. Trusted Boot can be divided into two phases:

- Boot of hardware platform: starts from the platform power on to the BIOS initialization and ends after the BIOS passes control rights to the boot loader.
- Startup of operating system: starts with the loading of operating system loader from the main boot sector, then the loading of the operating system kernel and ends with the running of the Init process.

The “measure” is a digest calculated with a cryptographic hash function on ehe entity to evaluate. This digest is stored in a special shielded location of the RTS in the TPM, the PCR. In order to simplify the evaluation of the stages of the platform from the boot to the runtime, a TPM has multiple PCRs, each one dedicated to store different measurements as Table 2.2 shows.

A TPM may maintain multiple banks of PCR. A PCR bank is a collection of PCR that are Extended with the same hash algorithm. PCR banks are identified by the hash algorithm used to Extend the PCR in that bank [7]. Multiple banks may handle situations where one hash algorithm is required for legacy or compatibility with one set of applications. Not all banks need to have the same number of PCR.

Trusted boot process based on TPM works as follows:

PCR Index	PCR Usage
0	SRTM, BIOS, Host Platform Extensions, Embedded Option, ROMs and PI Drivers
1	Host Platform Configuration
2	UEFI driver and application Code
3	UEFI driver and application Configuration and Data
4	UEFI Boot Manager Code (usually the MBR) and Boot Attempts
5	Boot Manager Code Configuration and Data (for use by the Boot Manager Code) and GPT/Partition Table
6	Host Platform Manufacturer Specific
7	Secure Boot Policy
8-15	Defined for use by the Static OS
16	Debug
23	Application Support

Table 2.2. PCR Usage [12]

1. Trusted BIOS loads boot loader and sends it to TPM to be measured and verified. Once TPM has verified its integrity, the boot program is loaded and the BIOS passes control rights to the CPU to run the Boot program.

2. TPM validates the operating system loader program, such as Grub in Linux. If the verification is successful, the Grub Stage 1 code is loaded and gains the trusted boot control.

3. The Grub Stage 1 validates Grub Stage 1.5 code with TPM, then loads and runs the code of the Stage 1.5 phase. At the end of this stage, the file system is mounted.

4. The Grub Stage 2 code is verified by TPM and loaded by trusted Grub Stage 1.5. Then, it verifies the integrity of the configuration file `/boot/Grub/Grub.conf` in which the locations of the disk partitions, the kernel image, and virtual RAM disk file `initrd` are recorded.

5. The Grub Stage 2 code opens the configuration file, reads the operating system kernel image, and tries to verify the integrity of the operating system kernel image by TPM. If it is successful, the operating system kernel image is loaded and gains control.

6. Once the operating system kernel image is loaded, TPM will measure and verify the Init process. If the Init process is trusted, the kernel key data structures will be created and the kernel Init process will be loaded and take control.

7. Firstly, the Init process determines the list of the kernel modules needed to be loaded and the daemons needed to be created based on the system configuration. Then, it

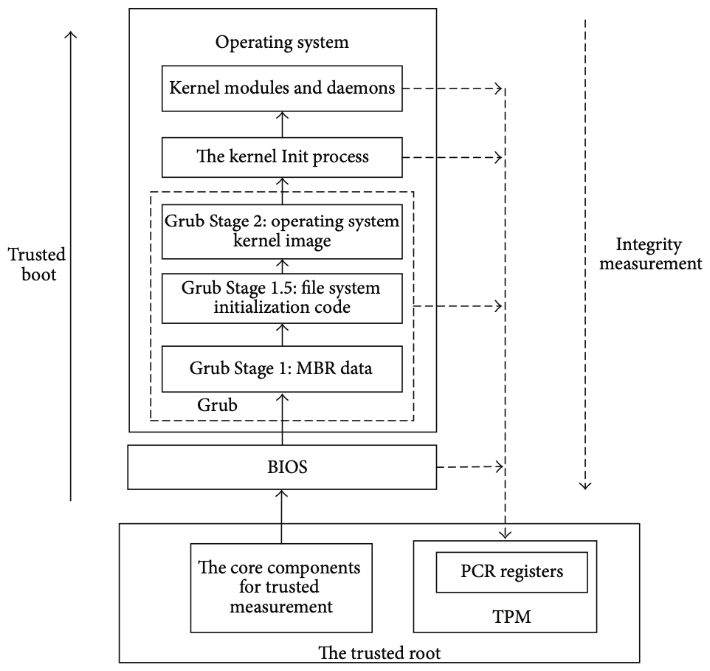

Operating system
Kernel modules and daemons
The kernel Init process
Grub Stage 2: operating system kernel image
Grub Stage 1.5: file system initialization code
Grub Stage 1: MBR data
Grub
BIOS
The core components for trusted measurement
PCR registers
TPM
The trusted root
Trusted boot
Integrity measurement

Figure 2.5. Trusted Boot in Linux [3]

will measure and verify each kernel module and daemon with TPM module before they are loaded. Only the trusted kernel modules and daemons are run sequentially to guarantee that the initialized computing environment is trusted. At last, the Init process starts receiving users' inputs, and a trusted computer is ready to be used.

2.3.3 TPM Identity

A Trusted Platform's state is reported by TPM using PCR values. An external entity that has to use this values in order to establish if platform behaves as expected, must check the identify the RTR (TPM) that provided the quote and the correct binding between RTR and the RTM.

TPM identification is possible thanks to embedded asymmetric keys called Endorsement Keys (EKs), derived from an endorsement seed contained in the TPM. TPM manufacturer provides the so-called Endorsement Certificate, a proof that Endorsement Key is generated from the endorsement seed belonging solely to that TPM. An external entity attests that the platform contains a Root-of-Trust-for Measurement, a genuine TPM, plus a trusted path between the RTM and the TPM through the Platform Certificate [7].

A direct use of Endorsement Key, since it is uniquely binded with a TPM, could result in a release of personal information. TCG in fact recommends not to use the Endorsement key directly, but only to decrypt certificates of other keys generated by the TPM itself, the so-called Attestation Keys (AKs or AIKs). Attestation Keys can only be used to sign the measurement data to guaranteeing its authenticity and integrity has generated (the digest calculated on the content of PCRs). Attestation CA (or Privacy CA) and has the purpose to certify that a given AK has been generated by a valid TPM.

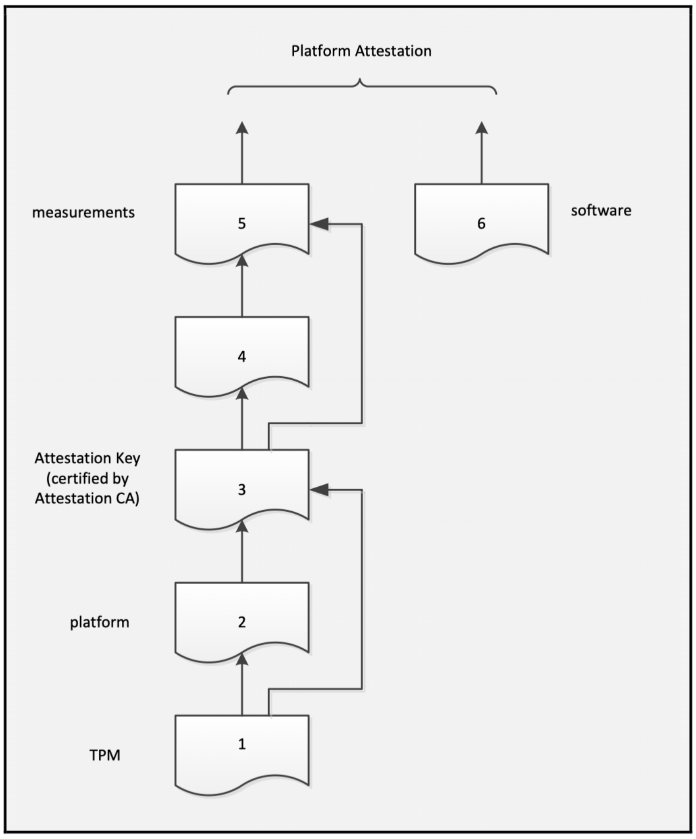


Figure 2.6. Attestation Hierarchy [10]

2.3.4 Attestation Hierarchy

Trusted platforms employ a hierarchy of attestations [10]:

1. Endorsement Certificate, provided by the TPM manufacturer, attests that the TPM is genuine and complies with this TPM specification
2. Platform Certificate, provided by the platform manufacturer, attests that the platform contains a Root-of-Trust-forMeasurement, a genuine TPM, plus a trusted path between the RTM and the TPM.
3. Attestation Key Certificate, provided by an Attestation CA, attests that an AK key has been generated by an unidentified but genuine TPM.
4. Trusted Platform provided the so-called Quote that attests a particular software/-firmware state. A quote takes the form of a signature over a software/firmware measurement in a PCR using an attestation key protected by the TPM.
5. Third-party Certification, issued by an external entity called Verifier, attests the correctness of the measurements of a Trusted Platform.

2.4 TPM Software Stack (TSS)

TPM Software Stack's aim is to isolate TPM programmers from low level detail of TPM architecture. TSS is composed by multiple layers (figure 2.7) and permits a scalable TPM

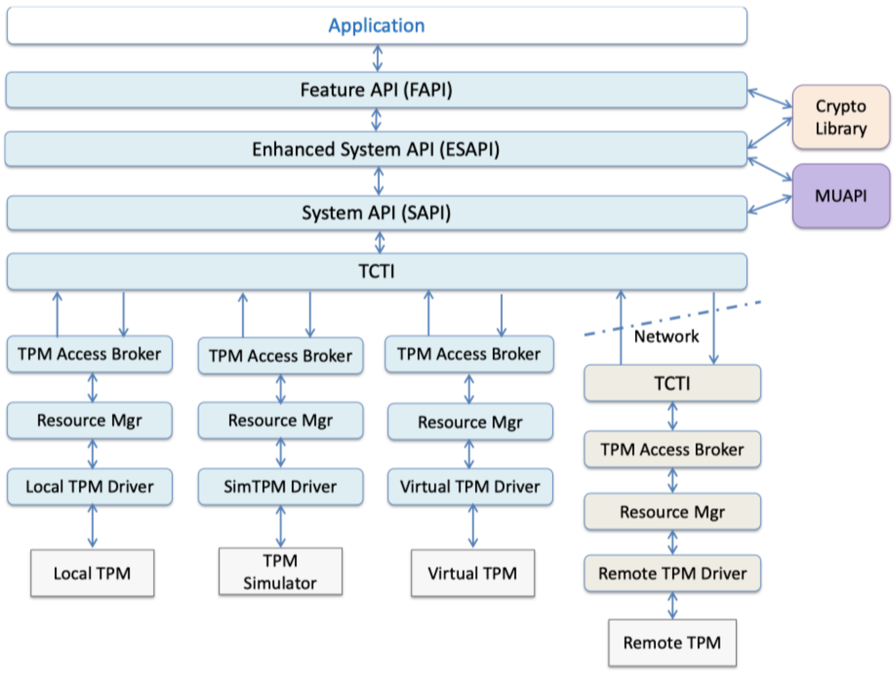


Figure 2.7. TPM Software Stack [18]

implementation both in high and low end systems [18]. TSS stack go from the application layer to the TPM in order of abstraction: from the Feature API which is the most high level to the TPM driver which is the most low level.

Feauture API (FAPI)

The Feature API (FAPI) provides the highest level of abstraction to developers. FAPI provides about 80% of the functionality of the TPM and with this level is possible to write an application without knowing any detail of the TPM low level architecture.

Enhanced System API (ESAPI)

The Enhanced System API (ESAPI) provides 100% of the functionality of the TPM. It reduces the complexity required of applications that desire to send individual “system level” TPM calls to the TPM, but that also require cryptographic operations on the data being passed to and from the TPM. ESAPI, unlike FAPI, requires a deep knowledge of how a TPM works.

System API (SAPI)

The System API (SAPI) is the TPM 2.0 equivalent of programming in the C language and gives access to all the functionalities of the TPM 2.0. It is designed for expert programmers who need to access directly to TPM: firmware, BIOS, OS, etc.

Marshaling/Unmarshaling (MUAPI)

The Marshaling/Unmarshaling API (MUAPI) performs the marshaling of TPM commands into byte streams and unmarshaling of the responses returned from the TPM.

TPM Command Transmission Interface (TCTI)

Through TPM Command Transmission Interface (TCTI) command byte streams are transmitted to the TPM and the application receives response byte streams from the TPM using the two most important TCTI functions, *transmit* and *receive*.

TPM Access Broker (TAB)

TPM Access Broker's (TAB) aim is to manage the processes concurrency for TPM applications and to ensure the completion of an operation without the interference of other processes.

Resource Manager

The resource manager handles the TPM context, similarly to OS's virtual memory manager. It performs objects swaps, sessions, and sequences in and out of the limited TPM onboard memory as needed. This layer is transparent to the upper layers of the TSS and is not mandatory. However, if not implemented, the upper layers will be responsible for TPM context management [18].

TPM Device Driver

TSS lower layer is the TPM Device Driver that receives a buffer of command bytes and a buffer length and performs the operations necessary to send those bytes to the TPM thanks to TPM I/O buffer.

Chapter 3

Linux IMA

The Trusted Computing Group (TCG) has defined a set of standards that describe how to take integrity measurements of a system and store the result in a separate trusted coprocessor (Trusted Platform Module) whose state cannot be compromised by a potentially malicious host system. This mechanism is called trusted boot.

IMA (Integrity Measurement Architecture) was introduced in Linux 2.6.30, as part of an overall Linux Integrity Subsystem. With IMA is possible to maintain the chain of trust measurements up to the application layer [2]. The goals of the kernel integrity subsystem are to detect if files have been accidentally or maliciously altered, both remotely and locally, appraise a file's measurement against a "good" value stored as an extended attribute, and enforce local file integrity [2].

Unlike the bootstrap process, an operating system handles a large variety of executable content (kernel, kernel modules, binaries. shared libraries, scripts, plugins) as shown in Figure 3.1. Furthermore, an operating system almost continuously loads executable content and measuring the content at each load time incurs a considerable performance overhead .

With IMA in order to let an external entity to attest the state of the platform during runtime, the measures collected cannot be only stored in a PCR otherwise the external entity could not be able to check if the measurement aggregate, contained in the PCR, represents or not a trusted state. Instead, a Measurement Log (ML) file is used in order to store the sequence of the Measurement Events (MEs) as they occurred in the system and a PCR in the TPM (typically PCR 10) is used to protect the integrity of this ML.

To prove to a remote party what software stack is loaded, the system needs to present the TPM state using the TCG attestation mechanisms and this Measurement Log. The remote party can then determine whether the ordered list has been tampered with and, once the list is validated, what kind of trust it associates with the measurements. Remote entity analyzes ML entry by entry, in order to determine if system is compromised.

3.1 Measuring Systems

The integrity of a program is a binary property that indicates whether the program and its environment have benne modified in an unauthorized manner. However integrity is a relative property that depends on the verifier's view of the ability of a program to protect itself. The IBM 4758 explicitly defines that the integrity of a program is determined by the code of the program and its ancestors [19]. For this reason, an essential feature

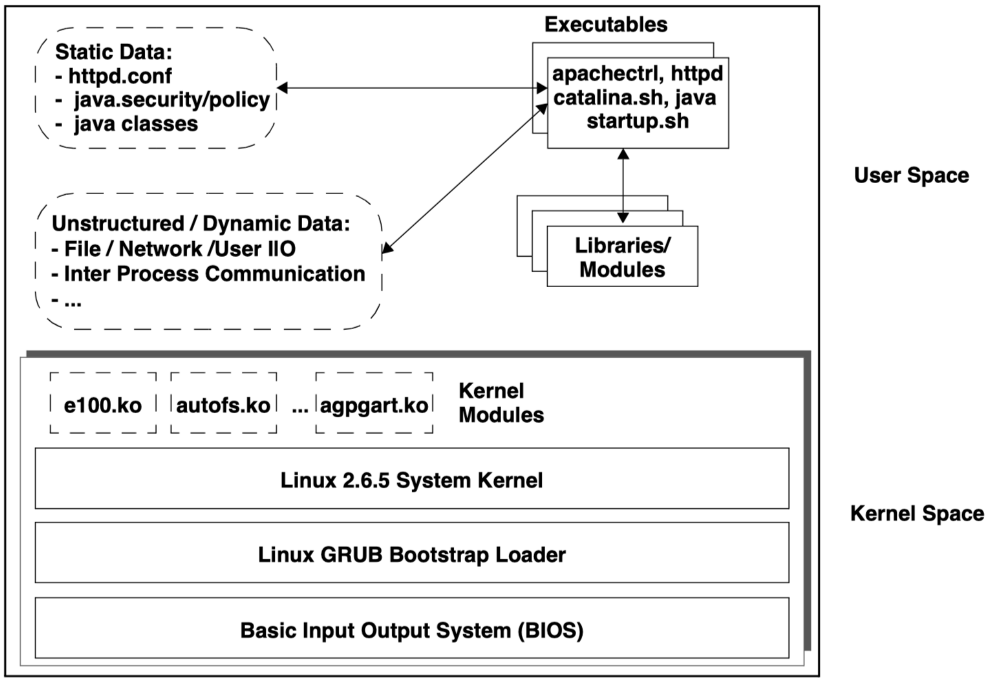


Figure 3.1. Runtime System Components [21]

that the system must support is the Trusted Boot in order to ensure that the verifier can understand if all the boot components are in the desired state. The application's integrity level depends on two types of data:

- High integrity data: data whose measure can be used by the verifier.
- Low integrity data: data whose measure cannot be used by the verifier.

In order to achieve Clark Wilson level of integrity verification [20] it is necessary to accomplish the following tasks:

- Verification Scope: the integrity of all processes must be measured. Otherwise, the scope of integrity impacting a process may be reduced to only those processes upon which it depends for high integrity code and data.
- Executable Content: For each process, all code executed must be of sufficient integrity regardless of whether it is loaded by the operating system, dynamic loader, or application.
- Structured data: data whose content has an identifiable integrity semantics and can be compared to reference values such as configuration files or security policy files. For each process, this type of data may be treated in the same manner as executable content above.
- Unstructured data: data whose content does not have an identifiable integrity semantics. the integrity of this type of data depends on the integrity of the processes that have modified it.

It is imperative for the Verifier to make sure that the measurement list is:

- Fresh and complete: list must include all measurements performed up to the time when the attestation is executed and it must not be subject to replay attacks.
- Unchanged: measurement are truly from the loaded executable and static data files and have not been tampered with.

IMA maintains a runtime measurement list, called Measurement Log (ML). The benefit of using TPM is that the measurement list cannot be compromised by any software attack, without being detectable. Hence, on a trusted boot system, IMA can be used to attest to the system's runtime integrity [2]. The goal of IMA is not to protect system's integrity, but it is at least to detect if such compromise has occurred, so that it can be repaired in a timely manner. A remote verifier can ensure that the list results in the value in PCR-10, and that the TPM has signed this value. With the use of TPM, a malicious software does not create a Measurement Log because it is not possible to reproduce TPM signature. IMA measurements are enabled with the kernel command line parameter:

```
ima_tcb=1
```

The IMA measurement list can be read through an IMA securityfs file, mounted at: `/sys/kernel/security/ima/ascii_runtime_measurements` and it looks like as following:

```
PCR   template-hash                           \
      filedata-hash                          filename-hint

10    7971593a7ad22a7cce5b234e4bc5d71b04696af4 ima \
      b5a166c10d153b7cc3e5b4f1eab1f71672b7c524 boot_aggregate
```

Figure 3.2. IMA measurement [2]

Chapter 4

Keylime

Keylime was presented in December 2016 with the whitepaper "Bootstrapping and Maintaining Trust in the Cloud" [22] by security research team in MIT's "Lincoln Laboratory". Keylime is a TPM-based highly scalable remote boot attestation and runtime integrity measurement solution [23]. It enables cloud users to monitor remote nodes using a hardware based cryptographic root of trust [24]. Keylime's allows to make TPM Technology easily accessible to developers and users without understanding TPM's lower levels operations. It also provides a flexible framework for the remote attestation of any given Platform Configuration Register.

4.1 Threat Model

Keylime's goal is to minimize trust in the cloud provider. Threat model assumes that the provider is at least "semi-trusted" [22]:

- The organization is trustworthy and the cloud provider has processes, technical controls, and policy in place to limit the impact of such compromise from spreading across their entire infrastructure.
- The attackers may be in control of some fraction of the cloud provider's resources. They can arbitrarily monitor or manipulate compromised portions of the cloud network or storage and they can not not physically tamper with any host's) CPU, bus, memory, or TPM (protection of physical components is not required).
- TPM and system manufacturers have created the appropriate endorsement credentials and have some mechanism to test their validity
- The attacker's goal is to obtain access to a tenant system to steal or deny the tenant's data or disturb services modifying the code at load-time or the process at run-time
- Load-time integrity measurement of the kernel can detect load-time modifications
- Runtime integrity measurement of the kernel can detect run-time modifications

4.2 Architecture

Keylime consists of four main components: Registrar, Verifier, Agent and Tenant [22].

4.2.1 Registrar

The Registrar stores and certifies the public AIKs of the TPMs and their Keylime UUIDs, which are indexes for the associated keys. The Registrar is only a trust root and does not store any tenant secrets. The tenant can decide to trust the registrar only after it attests its system integrity. Since the registrar is a simple a component with static code, verifying its integrity is very simple. Registrar can be hosted outside the cloud in the tenant's infrastructure or could be hosted on a physical system in the cloud.

4.2.2 Cloud Verifier

The Cloud Verifier (CV) is the core component of Keylime. Each cloud organization will have at least one CV that is responsible for verifying the system state of the organization's resources. As the Registrar it can be hosted in in the tenant's infrastructure or on a physical system in the cloud. The cloud verifier periodically polls (configurable frequency) agents for quotes and verifies them, to determine any violated policies. Once the Registrar has validated the node's AIK, it is possible to start the communication with Verifier.

4.2.3 Cloud Agent

The Cloud Agent runs in every cloud node and its duty it to send information about system state by sending quotes to Cloud Verifier.

4.2.4 Tenant

The Tenant is the Keylime interface which allows to issue commands to Cloud Agents and Verifier. It provides to the Agent an encrypted payload that contains information to start up his service and to the Cloud Verifier all policies to attest the integrity of nodes.

4.3 Framework

The Keylime framework is composed by two operational phases:

1. Physical Node Registration Protocol
2. Three Party Bootstrap Key Derivation Protocol

4.3.1 Physical Node Registration Protocol

The aim of node registration protocol, shown in Figure 4.1 is to validate the node's keys. Firstly the node sends to Keylime Registrar its ID and TPM credentials (AIK_{pub}, EK_{pub}). The Registrar, after checking the validity of the received EK with the TPM manufacturer, challenges the cloud node: it creates an ephemeral symmetric key K_e and encrypts it along with a hash of the AIK_{pub} with EK_{pub}. After receiving the Registrar's challenge, Cloud Node uses ActivateCredential TPM command to decrypt K_e with the corresponding private key parts of EK and AIK and proves its identity by sending to the Registrar the HMAC of its ID computed with K_e, so that the Registrar marks its AIK as valid.

The TPM primitives used during Node Registration Protocol are [25]:

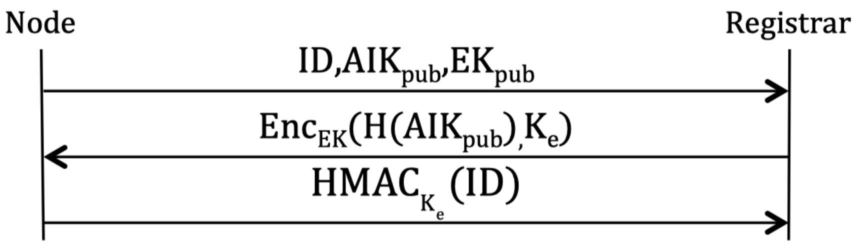


Figure 4.1. Physical node registration protocol [22]

- tpm2_startauthsession: starts a session with the TPM.
- tpm2_policysecret: couples the authorization of an object to that of an existing object.
- tpm2_activatecredential: enables access to the credential qualifier to recover the credential secret.
- tpm2_flushcontext: remove a specified handle, or all contexts associated with a transient object, loaded session or saved session from the TPM.

4.3.2 Three Party Bootstrap Key Derivation Protocol

The second step of Keylime Framework is the Three Party Bootstrap Key Derivation Protocol, whose aim is to deliver the bootstrap key K_b to Cloud Node. At the start of the process, the tenant generates a symmetric encryption key K_b and split it into two parts U and V (random value) such that $U = K_b \bigoplus V$. The Tenant sends U to the Cloud Node and shares V to the Verifier which will send it to the Cloud Node only after after a successful integrity verification. After obtaining the node UUID and IP address, the Tenant notifies the Verifier of their intent to boot a Cloud Node (area A in Figure 4.2). The Tenant connects to the CV over a secure channel, such as mutually authenticated TLS, and provides V, UUID, Node IP, and a TPM policy (it specifies a whitelist of acceptable PCR values to expect from the TPM of the cloud node). At this point, both Tenant and Verifier can now attest the node in parallel.

As Figure 4.2 shown, the attestation protocol is basically the same between the Verifier and the Cloud Node (B) and between of the Tenant and the Cloud Node (C). It consists of two groups of messages: the first for the initiator (either CV or tenant) to request a TPM quote and the second for the initiator to provide a share of K_b to the cloud node upon successful validation of the quote. Since Key Derivation Protocol's aim it to bootstrap keys into the system, there are no existing software keys with which we create a secure channel. Thus, this protocol must securely transmit a share of K_b over an untrusted network. This is accomplished by creating an asymmetric key pair on the node (NK) outside of the TPM. This uses 16th PCR value in a TPM quote to bind NK to the identity of the TPM, therefore authenticating NK. Verifier and tenant can encrypt its share of K_b using NK pub and deliver it to the node securely.

There is a little difference between the interaction with the Tenant and the Verifier: the Tenant only verifies identity of the node's AIK, whereas the Verifier also checks the integrity measures. In both interactions the protocols begins by sending a nonce (n_t for the Tenant n_{CV} for the Verifier) to the Node along with a mask indicating which PCRs it should include in its quote in reply (the Tenant sends an empty mask). The Cloud Node

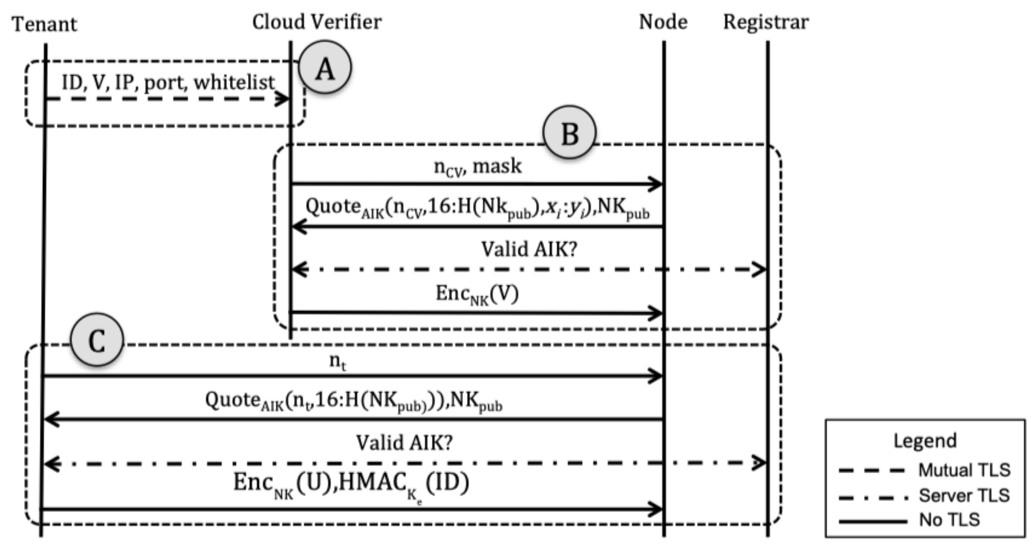


Figure 4.2. Three Party Bootstrap Key Derivation Protocol [22]

extends a hash of NK_{pub} into a blank PCR 16 and returns a quote (Quote AIK(nonce CV, 16 : H(NK_{pub}), x_i: y_i), where 16 : H(NK_{pub}) represents the PCR 16 containing the hash of NK_{pub}, while x_i: y_i represent the PCRs requested by the CV with their respective values (only for Verifier). At this point, the initiator has to confirm that the AIK is valid according to the Registrar. If the initiator is the Verifier, it has to check the other PCR values to ensure that they are valid according to the TPM policy [26].

The TPM primitives used during Three Party Bootstrap Key Derivation Protocol are [25]:

- tpm2_pcrreset: Reset one or more PCR banks. More than one PCR index can be specified. The reset value is manufacturer-dependent and is either sequence of 00 or FF on the length of the hash algorithm for each supported bank.
- tpm2_extend: Extends a PCR
- tpm2_quote: Provide quote and signature for given list of PCRs in given algorithm/banks
- tpm2_checkquote: Uses the public portion of the provided key to validate a quote generated by a TPM. This will validate the signature against the quote message and, if provided, verify that the qualifying data and PCR values match those in the quote.
- tpm2_nvdefine: Define an NV index with given auth value.
- tpm2_nvwrite: Write data specified via FILE to a Non-Volatile (NV) index.

4.4 Deployment

This section describes how to configure machines used for testing. Machines used for these testing purposes were:

- Attester Machine: Raspberry Pi 4 with Raspbian OS equipped with an Infineon TPM v2 running Keylime Agent
- Verifier Machine: Ubuntu Desktop 20.04 LTS with an Intel Core i5 1,4 GHz (4 cores, 8 threads) running Keylime Verifier, Keylime Registrar and Keylime Tenant Webapp

To determine whether a machine has a TPM in Linux and the driver for TPM been successfully loaded by the Linux kernel we can use:

```
$ dmesg | grep -i tpm
```

To install Keylime, we must first install the TPM2 software stack, tools and resource manager. Keylime also requires libtss2 with version >= 2.4.0, while by default Ubuntu 20.04 has libtss2 version 2.3.2.

```
$ sudo apt update
$ sudo apt install autoconf autoconf-archive git libglib2.0-dev \
  libtool pkg-config libjson-c-dev libcurl4-gnutls-dev
```

Install libtss2 library:

```
$ git clone https://github.com/tpm2-software/tpm2-tss.git
$ cd tpm2-tss
$ ./bootstrap
$ ./configure --prefix=/usr
$ make
$ sudo make install
$ cd ..
```

Install tpm2-tools:

```
$ git clone https://github.com/tpm2-software/tpm2-tools.git
$ cd tpm2-tools
$ ./bootstrap
$ ./configure --prefix=/usr/local
$ make
$ sudo make install
$ cd ..
```

Install TPM 2.0 Resource Manager:

```
$ git clone https://github.com/tpm2-software/tpm2-abrmd.git
$ cd tpm2-abrmd
$ git checkout tags/2.4.1
$ ./bootstrap
$ ./configure --with-dbuspolicydir=/etc/dbus-1/system.d \
        --with-systemdsystemunitdir=/lib/systemd/system \
        --with-systemdpresetdir=/lib/systemd/system-preset \
        --datarootdir=/usr/share
$ make
$ sudo make install
$ sudo ldconfig
$ cd ..
```

Add TPM 2.0 Resource Manager user:

```
$ sudo useradd --system --user-group tss
```

Start tpm2-abrmd service:

```
$ sudo pkill -HUP dbus-daemon
$ sudo systemctl daemon-reload
$ sudo systemctl enable tpm2-abrmd.service
$ sudo systemctl start tpm2-abrmd.service
```

Check that `tpm2-abrmd.service` is running:

```
$ sudo systemctl status tpm2-abrmd.service
```

Configure TPM Command Transmission Interface (TCTI) for Tabrmd:

```
$ export TPM2TOOLS_TCTI="tabrmd:bus_name=com.intel.tss2.Tabrmd"
```

Check tpm2tools are able to communicate with the TPM via the access broker resource manager

```
$ tpm2_pcrread
```

This should print out the current values of PCRs. For example:

```
sha1:
  0 : 0xde3e4800b622034da4ac6bb5c5c9cb0021daee7e
  1 : 0x3c4abe581750af1c2e8ca2ffe43a8ffb8cf138fa
  2 : 0x904d221b2e2a0abde122725a0575eb5d93285a16
  3 : 0xfca52eec12db4002acd59bf8adfafea17e33cbac
  4 : 0xa07f29ee7004c3abeb06ab514d11d1bf07dc5ca6
  5 : 0x394d1ee03aaaa745a4aa9f6d0016efce9f10089b
  6 : 0xa37180cbb51e652c3072a539c25dc5fee5a9997e
  7 : 0x0000000000000000000000000000000000000000
  8 : 0x0000000000000000000000000000000000000000
  9 : 0x0000000000000000000000000000000000000000
  10: 0x13de482eeebb9558f356d36eea51e9e440a8222a
  11: 0x0000000000000000000000000000000000000000
  12: 0x0000000000000000000000000000000000000000
  13: 0x0000000000000000000000000000000000000000
  14: 0x0000000000000000000000000000000000000000
  15: 0x0000000000000000000000000000000000000000
  16: 0x1957fad45f8eb90d2c2a818bfbc501a5c9ada108
  17: 0xffffffffffffffffffffffffffffffffffffffff
  18: 0xffffffffffffffffffffffffffffffffffffffff
  19: 0xffffffffffffffffffffffffffffffffffffffff
  20: 0xffffffffffffffffffffffffffffffffffffffff
  21: 0xffffffffffffffffffffffffffffffffffffffff
  22: 0xffffffffffffffffffffffffffffffffffffffff
  23: 0x0000000000000000000000000000000000000000
```

Keylime installation

```
$ git clone https://github.com/keylime/keylime.git
$ cd keylime
$ sudo pip3 install . -r requirements.txt
$ sudo cp keylime.conf /etc/
```

4.4.1 Verifier Machine

In order to perform proposed tests, it was used a unique machine running the Keylime Verifier, Keylime Registrar and Keylime Tenant components.

Keylime Registrar

Edit Keylime configuration file:

```
$ sudo nano /etc/keylime.conf
```

Inside [registrar] section modify only the registrar_ip parameter and set it with the IP address of the host, for example:

```
registrar_ip = 192.168.1.189
```

Keylime Verifier

Edit Keylime configuration file:

```
$ sudo nano /etc/keylime.conf
```

Inside [cloud_verifier] section, set cloudverifier_ip, registrar_ip, revocation_notifier_ip paramenters to the host IP address, for example:

```
cloudverifier_ip = 192.168.1.189
registrar_ip = 192.168.1.189
revocation_notifier_ip = 192.168.1.189
```

Keylime Tenant

Edit Keylime configuration file:

```
$ sudo nano /etc/keylime.conf
```

Inside [tenant] section, set cloudverifier_ip and registrar_ip, for example:

```
cloudverifier_ip = 192.168.1.189
registrar_ip = 192.168.1.189
```

Keylime Tenant use tpm_policy parameter of the configuration file to check the Trusted Boot. On the machine that requires remote attestation, read PCR registers values:

```
$ tpm2_pcrread
```

Write these values from PCR 0 to 7 of the SHA256 bank to the tpm_policy parameter inside [tenant] section, for example:

```
tpm_policy = {"0": ["E2E..."], ..., "6": [F8E...], "7": ["000..."]}
```

Before registering, Keylime Tenant checks that EK certificate sent by the Agent was issued by a CA whose certificate is stored in the directory `/var/lib/keylime/tpm_cert_store/`.

On the machine that requires remote attestation, look for CA issuer of the EK certificate:

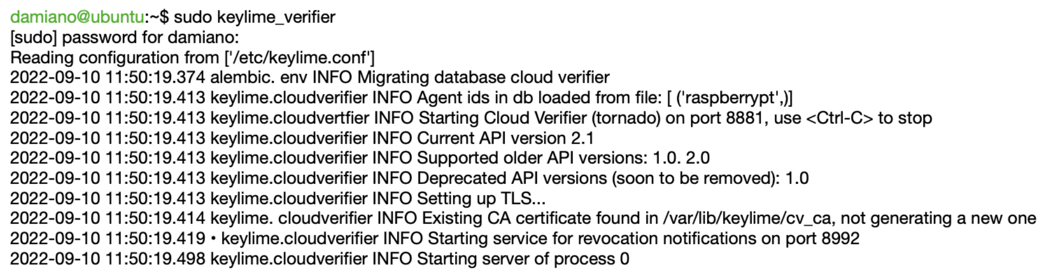
```
$ tpm2_getekcertificate -o RSA_EK_cert.bin -o ECC_EK_cert.bin
$ openssl x509 -inform der -in RSA_EK_cert.bin -noout -text
```

Find CA Issuers field, for example:

```
URI:http://pki.infinenon.com/OptigaRsaMfrCA040/OptigaRsaMfrCA040.crt
```

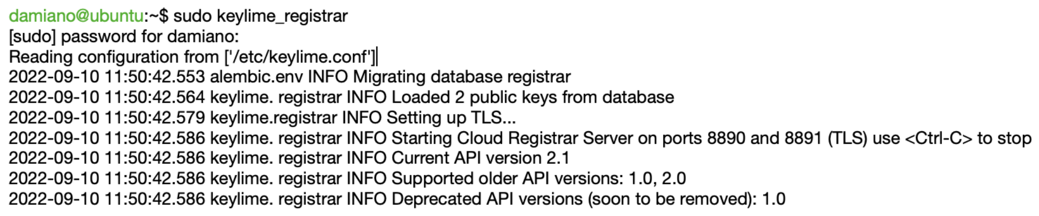
On the machine running Keylime Verifier:

```
$ curl http://pki.infineon.com/OptigaRsaMfrCA040/OptigaRsaMfrCA040.crt \
  -o OptigaRsaMfrCA040.crt
$  openssl x509 -inform der -in ./OptigaRsaMfrCA049.crt \
   -outform pem -out /var/lib/keylime/tpm_cert_store/OptigaRsaMfrCA040.pem
```



```
damiano@ubuntu:~$ sudo keylime_verifier
[sudo] password for damiano:
Reading configuration from ['/etc/keylime.conf']
2022-09-10 11:50:19.374 alembic. env INFO Migrating database cloud verifier
2022-09-10 11:50:19.413 keylime.cloudverifier INFO Agent ids in db loaded from file: [ ('raspberrypt',)]
2022-09-10 11:50:19.413 keylime.cloudvertfier INFO Starting Cloud Verifier (tornado) on port 8881, use <Ctrl-C> to stop
2022-09-10 11:50:19.413 keylime.cloudverifier INFO Current API version 2.1
2022-09-10 11:50:19.413 keylime.cloudverifier INFO Supported older API versions: 1.0. 2.0
2022-09-10 11:50:19.413 keylime.cloudverifier INFO Deprecated API versions (soon to be removed): 1.0
2022-09-10 11:50:19.413 keylime.cloudverifier INFO Setting up TLS...
2022-09-10 11:50:19.414 keylime. cloudverifier INFO Existing CA certificate found in /var/lib/keylime/cv_ca, not generating a new one
2022-09-10 11:50:19.419 • keylime.cloudverifier INFO Starting service for revocation notifications on port 8992
2022-09-10 11:50:19.498 keylime.cloudverifier INFO Starting server of process 0
```

Figure 4.3. Keylime Verifier Output



```
damiano@ubuntu:~$ sudo keylime_registrar
[sudo] password for damiano:
Reading configuration from ['/etc/keylime.conf']
2022-09-10 11:50:42.553 alembic.env INFO Migrating database registrar
2022-09-10 11:50:42.564 keylime. registrar INFO Loaded 2 public keys from database
2022-09-10 11:50:42.579 keylime.registrar INFO Setting up TLS...
2022-09-10 11:50:42.586 keylime. registrar INFO Starting Cloud Registrar Server on ports 8890 and 8891 (TLS) use <Ctrl-C> to stop
2022-09-10 11:50:42.586 keylime. registrar INFO Current API version 2.1
2022-09-10 11:50:42.586 keylime. registrar INFO Supported older API versions: 1.0, 2.0
2022-09-10 11:50:42.586 keylime. registrar INFO Deprecated API versions (soon to be removed): 1.0
```

Figure 4.4. Keylime Registrar Output

Installing Keylime Verifier and Registrar services

In order to install Keylime Verifier and Registrar as a systemd service, edit `installer.sh`:

```
$ sudo nano services/installer.sh
```

Comment out the lines relating to the Agent service:

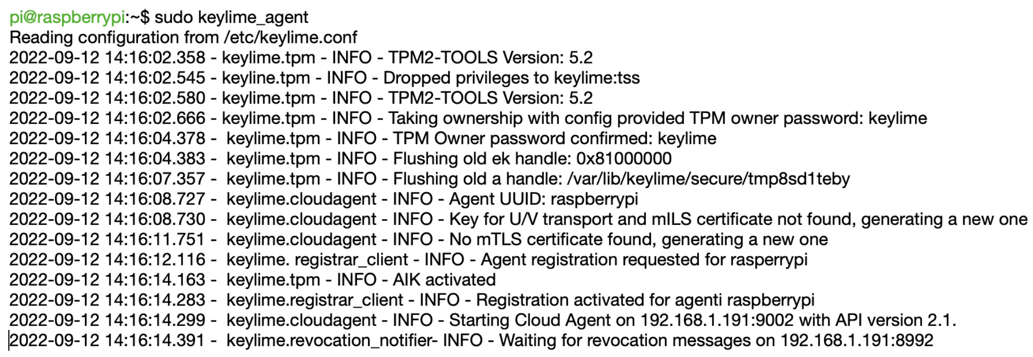
```
# prepare keylime service files and store them in systemd path
#sed "s|KEYLIMEDIR| $KEYLIMEDIR|g" $BASEDIR/keylime_agent.service.template >
/etc/systemd/system/keylime_agent.service
sed "s|KEYLIMEDIR| $KEYLIMEDIR|g" $BASEDIR/keylime_registrar.service.template >
/etc/systemd/system/keylime_registrar.service
sed "s|KEYLIMEDIR| $KEYLIMEDIR|g" $BASEDIR/keylime_verifier.service.template >
/etc/systemd/system/keylime_verifier.service

# set permissions
#chmod 664 /etc/systemd/system/keylime_agent.service
chmod 664 /etc/systemd/system/keylime_registrar.service
chmod 664 /etc/systemd/system/keylime_verifier.service

# enable at startup
#systemctl enable keylime_agent.service
systemctl enable keylime_registrar.service
systemctl enable keylime_verifier.service
```

Run the `installer.sh` script and start Keylime Verifier Service and Keylime Registrar Service:

```
$ sudo ./services/installer.sh
$ sudo systemctl start keylime_verifier.service
$ sudo systemctl start keylime_registrar.service
```

4.4.2 Keylime Agent

First of all it is necessary to edit Keylime configuration file:



```
pi@raspberrypi:~$ sudo keylime_agent
Reading configuration from /etc/keylime.conf
2022-09-12 14:16:02.358 - keylime.tpm - INFO - TPM2-TOOLS Version: 5.2
2022-09-12 14:16:02.545 - keyline.tpm - INFO - Dropped privileges to keylime:tss
2022-09-12 14:16:02.580 - keylime.tpm - INFO - TPM2-TOOLS Version: 5.2
2022-09-12 14:16:02.666 - keylime.tpm - INFO - Taking ownership with config provided TPM owner password: keylime
2022-09-12 14:16:04.378 -  keylime.tpm - INFO - TPM Owner password confirmed: keylime
2022-09-12 14:16:04.383 -  keylime.tpm - INFO - Flushing old ek handle: 0x81000000
2022-09-12 14:16:07.357 -  keylime.tpm - INFO - Flushing old a handle: /var/lib/keylime/secure/tmp8sd1teby
2022-09-12 14:16:08.727 -  keylime.cloudagent - INFO - Agent UUID: raspberrypi
2022-09-12 14:16:08.730 -  keylime.cloudagent - INFO - Key for U/V transport and mILS certificate not found, generating a new one
2022-09-12 14:16:11.751 -  keylime.cloudagent - INFO - No mTLS certificate found, generating a new one
2022-09-12 14:16:12.116 -  keylime. registrar_client - INFO - Agent registration requested for rasperrypi
2022-09-12 14:16:14.163 -  keylime.tpm - INFO - AIK activated
2022-09-12 14:16:14.283 -  keylime.registrar_client - INFO - Registration activated for agenti raspberrypi
2022-09-12 14:16:14.299 -  keylime.cloudagent - INFO - Starting Cloud Agent on 192.168.1.191:9002 with API version 2.1.
2022-09-12 14:16:14.391 -  keylime.revocation_notifier- INFO - Waiting for revocation messages on 192.168.1.191:8992
```

Figure 4.5. Keylime Agent Output

```
$ sudo nano /etc/keylime.conf
```

In the [general] section, assign the address of the machine running the Keylime Verifier to the receive_revocation_ip parameter, for example:

```
receive_revocation_ip = 192.168.1.189
```

In the [cloud agent] section, assign to cloud_agent_ip parameter the IP address of the host, for example:

```
cloud_agent_ip = 192.168.1.191
```

Assign to registrar_ip the IP address of the machine running Keylime Verifier (typically the same as Keylime Verifier), for example:

```
registrar_ip = 192.168.1.189
```

In order to install Keylime Agent as a systemd service, edit `installer.sh`:

```
$ sudo nano services/installer.sh
```

Comment out the lines relating to the Verifier and Registrar services:

```
# prepare keylime service files and store them in systemd path
sed "s|KEYLIMEDIR| $KEYLIMEDIR|g" $BASEDIR/keylime_agent.service.template >
/etc/systemd/system/keylime_agent.service
#sed "s|KEYLIMEDIR| $KEYLIMEDIR|g" $BASEDIR/keylime_registrar.service.template >
/etc/systemd/system/keylime_registrar.service
#sed "s|KEYLIMEDIR| $KEYLIMEDIR|g" $BASEDIR/keylime_verifier.service.template >
/etc/systemd/system/keylime_verifier.service

# set permissions
chmod 664 /etc/systemd/system/keylime_agent.service
#chmod 664 /etc/systemd/system/keylime_registrar.service
#chmod 664 /etc/systemd/system/keylime_verifier.service

# enable at startup
systemctl enable keylime_agent.service
#systemctl enable keylime_registrar.service
#systemctl enable keylime_verifier.service
```

Before starting Keylime Agent it is necessary to copy the file called `cacert.crt` from the directory `/var/lib/keylime/cv_ca/` of the node running Keylime Verifier to the same directory of the node running Keylime Agent. This file is the CA certificate of the Keylime Verifier and it is mandatory for the correct registration of the Keylime Agent. After that, it is possible to run the `installer.sh` script and start `keylime_agent.service`:

```
$ sudo ./services/installer.sh
$ sudo systemctl start keylime_agent.service
```

4.4.3 Keylime Webapp

Once the Verifier, Registrar and Agent have been installed and configured, they await commands issued by the Tenant. These commands can be issued via command line or via a provided web interface called Webapp:

```
$ sudo keylime_webapp
```

This command starts a web server which by default listens on port 443. Through the web interface it is possible to view a list of all registered Agents, register new ones or delete them. Once an Agent has been correctly registered, its information and the policies used for the attestation can be viewed with a simple click as shown in Figure 5.1.

Chapter 5

Testing

In this chapter will list the tests performed in order to verify the correct functioning of the Remote Attestation on Keylime Framework.

5.1 Testbed

The Testbed on which all the tests were performed was configured as follows:

- Verifier Machine: Ubuntu Desktop 20.04 LTS with an Intel Core i5 1.4 GHz (4 cores, 8 threads) running Keylime Verifier, Keylime Registrar and Keylime Tenant Webapp.
- Attester Machine: Raspberry Pi 4 running Keylime Agent.

5.1.1 Attester Machine

Raspberry 4 Pi Model B (4 GB of RAM) running Raspbian OS and equipped with Infineon Iridium TPM evaluation board (TPM 9670_Raspberry) and an OPTIGA SLI 9670AQ2.0.

Kernel and bootloader

We want to be able to use the TPM prior to booting the Linux kernel [27]. To do that, we need to add a second-stage bootloader (u-boot in our case) [28] with TPM support: Get the Raspbian image.

```
$ wget -O raspian_latest.zip https://downloads.raspberrypi.org/
  raspbian_lite_latest
$ unzip raspbian_lastest.zip
```

Flash the Raspbian image onto the card:

```
$ sudo dd if=2020-02-13-raspbian-buster-lite.img of=/dev/mmcblk0 bs=4M
  status=progress conv=fsync
```

There should be two partitions on the SD card now, `boot` and `rootfs`. Check if your toolchain is working:

```
$ arm-linux-gnueabihf-gcc --version
```

Get the Kernel Sources:

```
$ git clone https://github.com/raspberrypi/linux
```

Build the kernel:

```
$ cd linux
$ KERNEL=kernel71
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2711_defconfig
```

The menuconfig tool requires the ncurses development headers to compile properly:

```
$ sudo apt install libncurses5-dev
```

Compile and run the menuconfig utility as follows:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
```

Enable the following settings [29]:

```
Security options
        (y) Enable different security models
        (y) Integrity subsystem
        (y) Integrity Measurement Architecture(IMA)
        (y) Enable multiple writes to the IMA policy
        (y) Enable reading back the current IMA policy

Device Drivers
        (y) SPi Support
                (y) BCM2835 SPI controller
        TPM Hardware Support
                (y) TPM 2.0 FIFO Interface
```

The default hash algorithm is SHA-1. Remove the following line from drivers/clk/bcm/clk-bcm2835.c:

```
pesteore_initcall(bem2835_elk_driver_init);
```

Replace it with:

```
subsys_initcall(__bcm2835_clk_driver_init);
```

Make sure the root partition on your SD card is mounted (`/run/media/SD_NAME/rootfs`). Next, install the kernel modules onto the SD card:

```
$ sudo env PATH=$PATH make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
  INSTALL_MOD_PATH=/run/media/SD_NAME/rootfs modules_install
$ KERNEL=kernel71.img
$ sudo cp /run/media/SD_NAME/boot/$KERNEL /run/media/SD_NAME/boot/
  $KERNEL.bak
$ sudo cp result/arch/arm/boot/Image /run/media/SD_NAME/boot/$KERNEL
$ sudo cp result/arch/arm/boot/dts/broadcom/*.dtb /run/media/SD_NAME/boot/
$ sudo cp result/arch/arm/boot/dts/overlays/*.dtb* /run/media/SD_NAME/
  boot/overlays/
```

Setting up U-boot:

```
$ git clone https://gitlab.denx.de/u-boot/u-boot.git
$ cd u-boot
$ make -j$(nproc) CROSS_COMPILE=arm-linux-gnueabihf- rpi_4_defconfig
```

The configuration is saved in `.config` and it is necessary to change some parameters:

```
$ make menuconfig
```

Navigate through the menu and enable (y) in this order:

```
Device Drivers
        (y) SPI Support
                (y) Enable Driver Model for SPI drivers (DM_SPI)
                (y) Soft SPI driver (SOFT_SPI)
Library routines
        Security Support
                (y) Trusted Platform Module (TPM) support (TPM)
Device Drivers
        TPM support
                (y) TPMv2.x support (TPM_V2)
                (y) Enable support for TPMv2.x SPI chips (TPM2_TIS_SPI)
Command line interface
        Security commands
                (y) Enable the tpm command (CMD_TPM)
```

Save and exit the menu. Now you can build U-Boot:

```
$ make -j$(nproc) CROSS_COMPILE=arm-linux-gnueabihf- all
```

The most important result is `u-boot.bin`, our second stage bootloader. However it is necessary to create a script named `boot.scr` which specifies:

- The kernel which is to be booted by U-Boot (`kernel7l.img`)
- Send a clear command to the TPM to clear the 3 hierarchy authorization values
- Extend PCR as Table 5.1 shown

Copy this into a file named `boot.scr`:

```
tpm2 startup TPM2_SU_CLEAR

fatload mmc 0 ${loadaddr} u-boot.bin
hash sha256 ${loadaddr} ${filesize} *0x1000000
tpm2 pcr_extend 0 0x1000000

fatload mmc 0 ${loadaddr} boot.scr
hash sha256 ${loadaddr} ${filesize} *0x1000000
tpm2 pcr_extend 1 0x1000000

fatload mmc 0 ${kernel_addr_r} kernel7l.img
hash sha256 ${kernel_addr_r} ${filesize} *0x1000000
tpm2 pcr_extend 4 0x1000000

fatload mmc 0 ${loadaddr} config.txt
hash sha256 ${loadaddr} ${filesize} *0x1000000
tpm2 pcr_extend 5 0x1000000

fatload mmc 0 ${loadaddr} cmdline.txt
hash sha256 ${loadaddr} ${filesize} *0x1000000
tpm2 pcr_extend 6 0x1000000

setenv fdt_addr_r 0x03000000
fdt addr ${fdt_addr_r} && fdt get value bootargs /chosen bootargs

booti ${kernel_addr_r} - ${fdt_addr_r}
```

Convert this script into a binary format:

```
$ ./tools/mkimage -A arm64 -T script -C none -n "Boot script" -d boot.scr
  boot.scr.uimg
```

PCR Index	**Target Measurement(s)**	**Actor**
0	`u-boot.bin` (u-Boot image)	u-Boot
1	`boot.scr` (boot script file)	u-Boot
2	`boot.env` (environment file)	u-Boot
3	`platform.dtb` (binary data that describes platform's hardware)	u-Boot
4	`zImage/uImage` (Linux kernel image)	u-Boot
5	Platform configuration file (e.g. `config.txt`)	u-Boot
6	Additional parameters for Linux kernel command line (e.g. `cmdline.txt`)	u-Boot
7	`uInitrd` (Initial RAM disk image)	u-Boot
8	Operating System	Linux systemd service
9	TC endpoint PK Certificate (Trusted Channel binding)	Trusted Platform Agent
10	IMA Log checksum	Linux Kernel (IMA)

Table 5.1. List of PCRs and measured files

U-Boot needs a description of the hardware (information contained in the device tree `bcm2711-rpi-4-b.dtb` on the Raspberry Pi). To make changes to the device tree, we create a device tree overlay named `tpm-soft-spi.dts` and copy the following into it:

```
/*
 * Device Tree overlay for the Infineon SLB9670 Trusted Platform Module add-on
 * boards, which can be used as a secure key storage and hwrng.
 */

/dts-v1/;
/plugin/;

/ {
compatible = "brcm,bcm2835", "brcm,bcm2708", "brcm,bcm2709";

 fragment@0 {
  target = <&spi0>;
  __overlay__ {
    compatible = "spi-gpio";
    pinctrl-names = "default";
    pinctrl-0 = <&spi0_gpio7>;
    gpio-sck = <&gpio 11 0>;
    gpio-mosi = <&gpio 10 0>;
    gpio-miso = <&gpio 9 0>;
    cs-gpios = <&gpio 7 1>;
    spi-delay-us = <0>;
    #address-cells = <1>;
    #size-cells = <0>;
    status = "okay";

    /* for kernel driver */
    sck-gpios = <&gpio 11 0>;
    mosi-gpios = <&gpio 10 0>;
    miso-gpios = <&gpio 9 0>;
    num-chipselects = <1>;

    slb9670: slb9670@0 {
    compatible = "infineon,slb9670", "tis,tpm2-spi", "tcg,tpm_tis-spi";
    reg = <0>;
    gpio-reset = <&gpio 24 1>;
    #address-cells = <1>;
    #size-cells = <0>;
    status = "okay";

    /* for kernel driver */
    spi-max-frequency = <1000000>;
    };
  };
};

 fragment@1 {
  target = <&spi0_gpio7>;
  __overlay__ {
        brcm,pins = <7 8 9 10 11 24>;
        brcm,function = <0>;
  };
 };

fragment@2 {
  target = <&spidev0>;
  __overlay__ {
    status = "disabled";
  };
 };

 fragment@3 {
  target = <&spidev1>;
  __overlay__ {
    status = "disabled";
  };
 };
};
```

This file needs to be compiled into binary format using the device tree compiler dtc:

```
$ dtc -O dtb -b 0 -@ tpm-soft-spi.dts -o tpm-soft-spi.dtbo
```

Copy all U-Boot-related files onto the SD card:

```
$ cp u-boot.bin /run/media/SD_NAME/boot/
$ cp boot.scr.uimg /run/media/SD_NAME/boot/
$ cp tpm-soft-spi.dtbo /run/media/SD_NAME/boot/overlays
```

instruct the Raspberry's first-stage bootloader to use our TPM device tree overlay and load U-Boot instead of the Linux kernel. Edit `config.txt` adding the following lines:

```
dtparam=spi=on
dtoverlay=tpm-slb9670
dtoverlay=tpm-soft-spi
enable_uart=1
kernel=u-boot.bin
```

Finally Unmount the SD card:

```
$ sudo umount /run/media/SD_NAME/boot
$ sudo umount /run/media/SD_NAME/rootfs
```

Insert the flashed SD card and boot the Raspberry Pi. Open the file config. txt in an editor:

```
$ sudo nano /boot/cmdline.txt
```

Append the following to the existing line:

```
ima_policy=tcb
```

This policy measures all executables run, all mmap'd files for execution (such as shared libraries), all kernel modules loaded, and all firmware loaded. Additionally, all files read by root are measured as well. Reboot the Raspberry Pi and check if TPM is activated by:

```
$ ls /dev grep tpm
tpm0
tpmrm0
```

At this point it is possible to install Keylime and all softwares required as seen in the section 4.4.

Software installed and configured

- u-Boot boot loader (version 2020.04)
- Raspbian Buster Lite (kernel 4.19.118, no GUI)
- TPM2 TSS v3.2.0
- TPM2 TABRM v2.4.1
- TPM2 Tools v5.3
- TPM2 TSS Engine v1.1.0
- Cryptsetup with TPM support v2.0.3
- Keylime 6.5.0

Secure boot Configuration

- `/dev/mmblk0p1`: `/boot` (default)
- `/dev/mmblk0p2`: `/` (default)

- `/dev/mmblk0p3`: formatted ext4, not mounted, not used
- `/dev/mmblk0p4`: `/dev/mapper/EncSecPart` → `/mnt/encrypted`

The fourth partition (`/dev/mmblk0p4`) is encrypted with LUKS2 (Linux Unified Key Setup 2) format; the encryption key is stored in the TPM NV storage and protected by the sealing against PCRs 0-9 and with a known password stored on file. If any file measured by u-boot is modified, then the at reboot the encryption key cannot be unsealed and the partition mounted: therefore, the bootstrap process stops before its end and it is not possible to log in again. Those files must be changed in between a procedure of unsealing and resealing of the encryption key. This way, the secondary encrypted partition implements a form of secure boot.

5.2 Keylime Performance

In this section we evaluate the primary operations done by the TPM and keylime during verification protocol for Keylime Agents.

5.2.1 Preliminary Operations

In order to create a quote it is necessary to generate an AIK, since the EK can not be used for signing operations. Therefore the first thing to do is to generate a primary object from the primary seed. The command tpm2_create primary generates a primary RSA key of 2048 bit in the endorsement hierarchy:

```
$ tpm2_createprimary --hierarchy=e --hierarchy-auth=keylime \
  --hash-algorithm=sha256 --key-algorithm=rsa \
  --key-context=primary.ctx
```

At this point it is necessary to create a child object which can be used to sign the quotes. The command tpm2_create generates a child key and save saves the public and private portions inside the files provided:

```
$ tpm2_create --parent-context=primary.ctx \
  --hash-algorithm=sha256 --key-algorithm=rsa \
  --public=public.key --private=private.key
```

With the command tpm2_load this child object is loaded into TPM:

```
$ tpm2_load --parent-context=primary.ctx \
  --public=public.key --private=private.key \
  --key-context=child.ctx
```

With the command tpm2_readpublic it is possible to convert the public portion of the key to PEM format:

```
$ tpm2_readpublic --object -context=child.ctx \
  --format=pem --output=pubkey.pem
```

5.2.2 Verification Protocol

The time elapsed between the request for the Quote by the Verifier and the check of the same was measured. The following commands are executed during this protocol:

- tpm2_pcrreset: resets all PCR banks
- tpm2_pcrextend: performs the extend operation
- tpm2_quote: creates an output file with contains data signed by TPM. Qualification option, a nonce against replay attack, is provided by the verifier for each request for a quote.

  ```
  $ tpm2_quote --key-context=primary.ctx \
    --pcr-list=sha1 :0,1,2,3,4,5,6,7 --qualification=123456 \
    --message=msg --signature=sign \
    --hash-algorithm=sha256 --pcr=pcrs
  ```

- tpm2_checkquote: returns an output based on whether the quote is valid or not.

  ```
  $ tpm2_checkquote --public=pubkey.pem \
   --message=msg --signature=sign --pcr=pcrs \
   --hash-algorithm=sha256 --qualification =123456
  ```

All tests were performed on local platforms in order to minimize network latency and obtain the most accurate measurements of the protocol. Verification protocol script were run for a thousand times and then averaged the observed performance.

Verification Protocol	Create Quote	Check Quote	Network Latency
809.51 ms	773.337 ms	4.896 ms	31.277 ms

Table 5.2. Average Verification Protocol latency (ms)

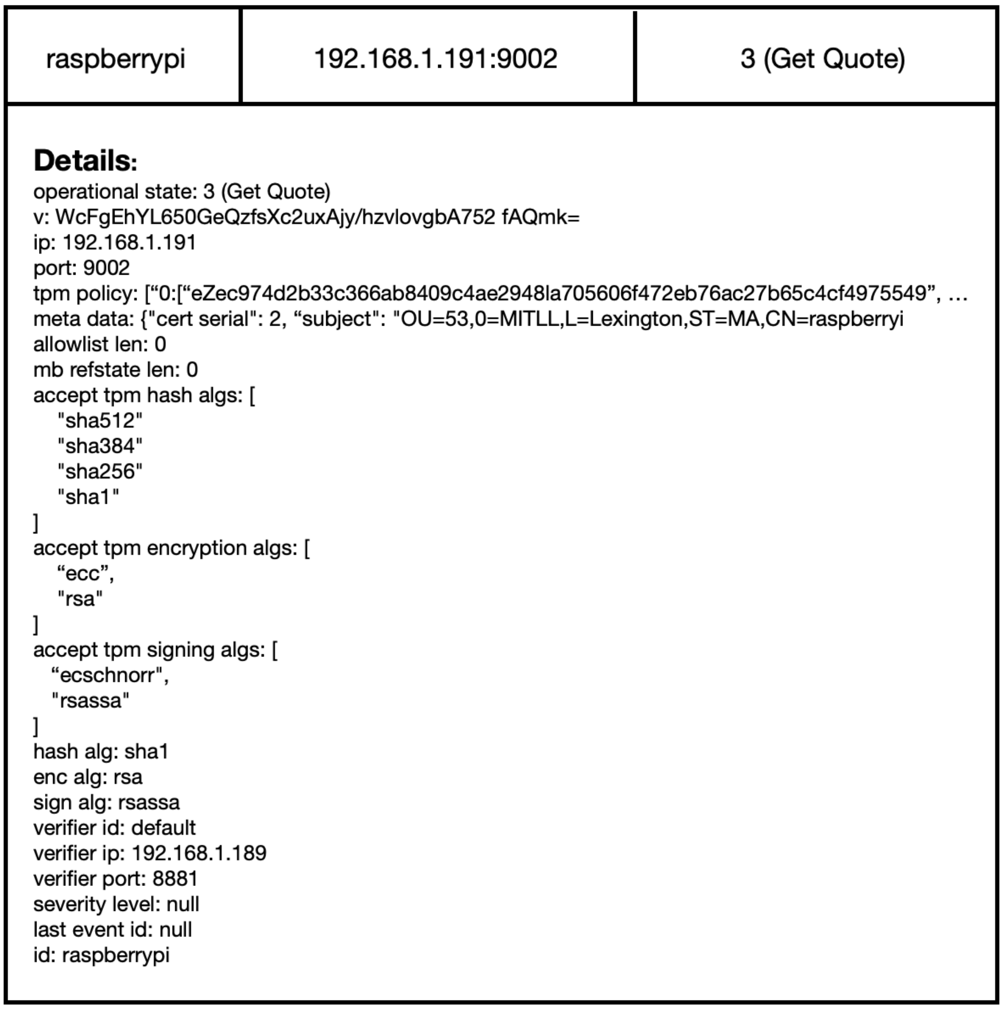
5.3 Functional tests

Functional tests goal is to verify if the behavior of the attestation process meets the requirements. In order to correctly register our Keylime Agent it is necessary to execute the following command:

```
$ sudo keylime_tenant --command add --targethost 192.168.1.191 \
  --cv 192.168.1.189 --uuid raspberrypi --cert default \
  --allowlist ./measurements.txt -- exclude excludelist.txt
```

Where:

- Target Host parameter is the IP address of the Keylime Agent
- cv parameter is the IP address of the Keylime Verifier
- UUID parameter is the unique identifier that you want to associate with the Keylime Agent
- Allow list parameter is a text file that contains the IMA measurement list
- Exclude list parameter is a text file that contains all files considered low integrity data (data whose measure cannot be used by the verifier)



raspberrypi	192.168.1.191:9002	3 (Get Quote)

Details:

```
operational state: 3 (Get Quote)
v: WcFgEhYL650GeQzfsXc2uxAjy/hzvlovgbA752 fAQmk=
ip: 192.168.1.191
port: 9002
tpm policy: [“0:[“eZec974d2b33c366ab8409c4ae2948la705606f472eb76ac27b65c4cf4975549”, …
meta data: {"cert serial": 2, “subject": "OU=53,0=MITLL,L=Lexington,ST=MA,CN=raspberryi
allowlist len: 0
mb refstate len: 0
accept tpm hash algs: [
    "sha512"
    "sha384"
    "sha256"
    "sha1"
]
accept tpm encryption algs: [
    “ecc”,
    "rsa"
]
accept tpm signing algs: [
   “ecschnorr",
   "rsassa"
]
hash alg: sha1
enc alg: rsa
sign alg: rsassa
verifier id: default
verifier ip: 192.168.1.189
verifier port: 8881
severity level: null
last event id: null
id: raspberrypi
```

Figure 5.1. Keylime Webapp

Once the Agent was correctly registered (Figure 5.1, it was possible to start running the tests. Assuming you know the SSH password of the machine running the Keylime Agent, once logged in, you tried to:

1. Edit a configuration file
2. Download an unexpected software from the Internet

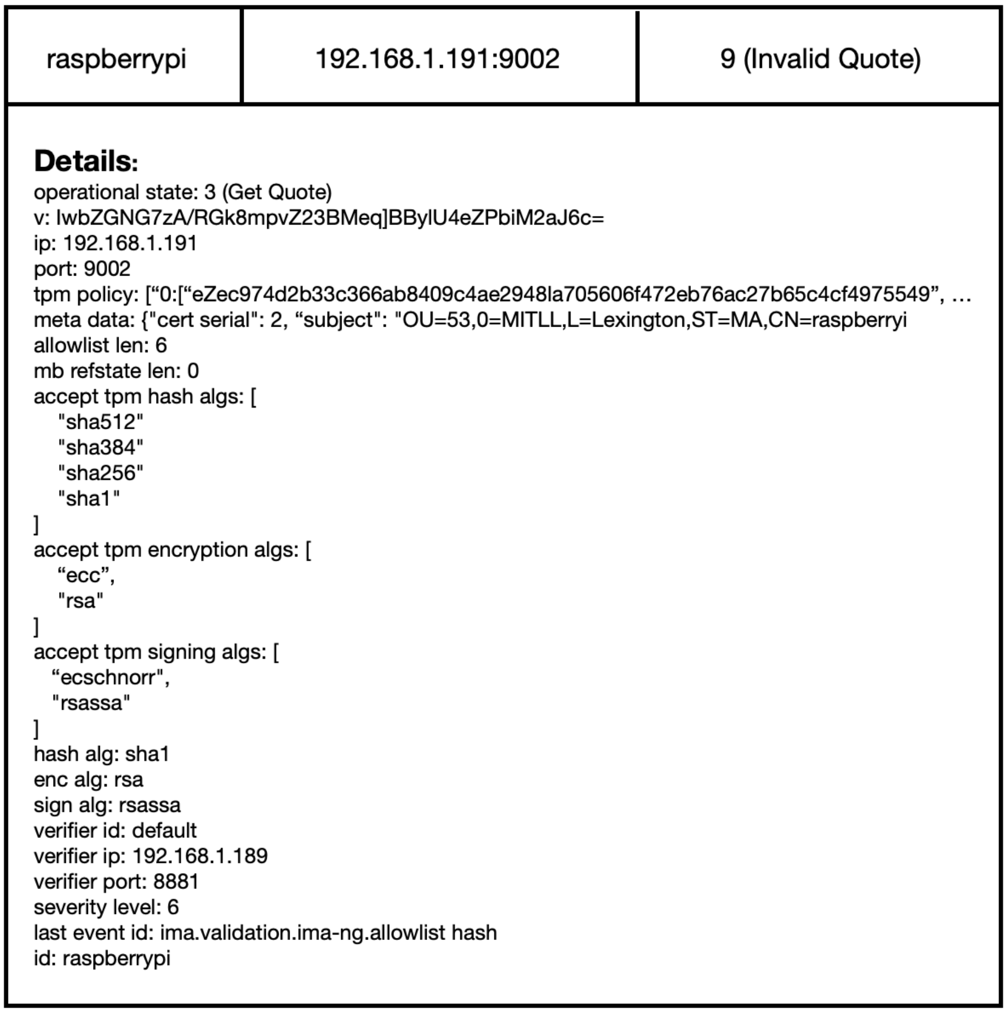
A new SSH connection will update the following files and this leads to the birth of these two errors:

- `/etc/hosts.allow`: lists the rules that allow access.

  ```
  2022-09-12 05:47:26,031 keylime.ima File not found in allowlist:
  etc/hosts.allow
  ```

- `/etc/hosts.deny`: lists the rules that deny access.

  ```
  2022-09-12 05:47:26,031 keylime.ima File not found in allowlist:
  etc/hosts.deny
  ```



raspberrypi	192.168.1.191:9002	9 (Invalid Quote)

Details:

```
operational state: 3 (Get Quote)
v: lwbZGNG7zA/RGk8mpvZ23BMeq]BBylU4eZPbiM2aJ6c=
ip: 192.168.1.191
port: 9002
tpm policy: [“0:[“eZec974d2b33c366ab8409c4ae2948la705606f472eb76ac27b65c4cf4975549”, ...
meta data: {"cert serial": 2, “subject": "OU=53,0=MITLL,L=Lexington,ST=MA,CN=raspberryi
allowlist len: 6
mb refstate len: 0
accept tpm hash algs: [
    "sha512"
    "sha384"
    "sha256"
    "sha1"
]
accept tpm encryption algs: [
    “ecc”,
    "rsa"
]
accept tpm signing algs: [
   “ecschnorr",
   "rsassa"
]
hash alg: sha1
enc alg: rsa
sign alg: rsassa
verifier id: default
verifier ip: 192.168.1.189
verifier port: 8881
severity level: 6
last event id: ima.validation.ima-ng.allowlist hash
id: raspberrypi
```

Figure 5.2. Keylime Webapp Invalid Quote

Even if you want to exclude these two files from the attestation process considering them as low integrity data, the configuration file (in our case xattr.conf) modification leads to the following error:

```
2022-09-12 05:47:26,031 keylime.ima WARNING File not found in allowlist:
etc/xattr.conf
```

The second case, the download of an unexpected software (in our case nmap) leads to the updating of numerous files inside the machine running the Keylime Agent and, consequently, to a disproportionate number of errors (only a few will be listed here):

```
2022-09-12 08:27:54,588 keylime.ima File not found in allowlist:
/etc/apt/apt.conf.d/01autoremove
2022-09-12 08:27:54,590 keylime.ima File not found in allowlist:
/etc/apt/apt.conf.d/20listchanges
2022-09-12 08:27:54,590 keylime.ima File not found in allowlist:
/usr/share/dpkg/cputable
2022-09-12 08:27:54,591 keylime.ima File not found in allowlist:
/etc/apt/sources.list
2022-09-12 08:27:54,591 keylime.ima File not found in allowlist:
/etc/dpkg/dpkg.cfg
2022-09-12 08:27:54,593 keylime.ima File not found in allowlist:
/etc/apt/sources.list.d/raspi.list
```

```
2022-09-12 08:27:54,593 keylime.ima File not found in allowlist:
/var/lib/apt/extended_states
2022-09-12 08:27:54,595 keylime.ima File not found in allowlist:
/var/lib/dpkg/status
.
.
.
2022-09-12 08:31:46,540 keylime.ima ERROR IMA ERRORS: Some entries couldn't
be validated. Number of failures in modes: Ima 900.
2022-09-12 08:31:46,614 keylime.cloudverifier WARNING Agent raspberrypi failed,
stopping polling
2022-09-12 08:31:46,807 keylime.revocation_notifier INFO Sending revocation event
to listening nodes...
```

Any one of these events leads to the update of the Attester Machine status to status 9 (Invalid Quote) as shown in Figure 5.2.

Chapter 6

Conclusion

Attestation is an area that will see many technological innovations and developments in the near future because the construction of a platform whose status can be monitored is essential.

Inside this project, firstly trusted boot on Raspberry Pi 4 has been enabled using Infineon TPM 2.0 chip. An open source standard developed by Trusted Computing Group (TCG), named TSS (TPM Software Stack) 2.0, was used to use the services offered by TPM 2.0 in conjunction with Raspbian OS, the operative system installed on Raspberry Pi. Linux IMA was activated in order to maintain the chain of trust measurements up to the application layer.

Configured the TPM and Linux IMA, the second part of the thesis introduces Keylime, an open-source tool for bootstrapping and maintaining trust in the cloud, its design and protocols. The fourth chapter describes how to install, configure and run keylime as well as the platforms (Attester Machine and Verifier Machine) on which we performed the evaluation.

The fifth chapter of the thesis consist of a series of tests for the TPM operations used by Keylime protocols. The TPM primitives used by Keylime during the Verification Protocol were analyzed and replicated in order to test the performance of the software. The behavior of the Remote Attestation was evaluated by trying to change the status of the Attester Machine by installing an unexpected software and modifying configuration files via the SSH protocol.

The main objective of this thesis is to demonstrate the advantages in terms of security that the use of a TPM brings. Thanks to the use of the Remote Attestation it is possible to to remotely detect adversarial presence on untrusted devices in order to guarantee their trustworthiness.

During the last few years we have seen a widespread diffusion of IoT devices which have now become an integral part of our lives (e.g. digital transportation, smart home, smart cities, wearable devices). According to Statista, the global market for Internet of things (IoT) end-user solutions is expected to grow to around 1.6 trillion U.S. dollars by 2025 [30]. However, as these devices perform safety-critical operations and contain sensitive information, they are increasingly being targeted to perform malicious exploitations.

Within this thesis it has been demonstrated that it is possible to perform the Remote Attestation with Linux Integrity Measurement Architecture enabled within a very limited environment such as a Raspberry Pi 4. Remote Attestation can therefore be a fundamental security technique to protect such IoT devices, allowing a trusted entity to that allows a trusted party (e.g. the Verifier) to assure the integrity of the untrusted IoT device (e.g. the Agents).

Bibliography

[1] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation", International Journal of Information Security, vol. 10, no. 2, 2011, pp. 63–81, DOI 10.1007/s10207-011-0124-7

[2] "Integrity Measurement Architecture (IMA)." `https://sourceforge.net/p/linux-ima/wiki/Home/`

[3] C. Huang, C. Hou, H. Dai, Y. Ding, S. Fu, and M. Ji, "Research on linux trusted boot method based on reverse integrity verification", Scientific Programming, vol. 2016, 2016, pp. 1–12, DOI 10.1155/2016/4516596

[4] L. Qiu, Y. Zhang, F. Wang, M. Kyung, and H. R. Mahajan, "Trusted computer system evaluation criteria", National Computer Security Center, 1985, DOI 10.1007/978-1-349-12020-8_1

[5] P. S. Tasker, "Trusted computer systems", 1981 IEEE Symposium on Security and Privacy, 1981, DOI 10.1109/SP.1981.10020

[6] "Trusted Platform Module (TPM) 2.0: A Brief Introduction." `https://www.trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-A-Brief-Introduction.pdf`

[7] Trusted Computing Group, "Trusted platform module library part 1: Architecture", TCG Published, October 30, 2014

[8] Trusted Computing Platform Alliance, "Main Specification Version 1.1b", February 22, 2002

[9] Trusted Computing Group, "TPM Main Part 1 Design Principles", TCG Published, March 1, 2011

[10] Trusted Computing Group, "Trusted Platform Module Library Part 1: Architecture", TCG Published, November 8, 2019

[11] "Trusted Platform Module Technology Overview." `https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/trusted-platform-module-overview`

[12] W. Arthur, D. Challener, and K. Goldman, "A practical guide to tpm 2.0: Using the new trusted platform module in the new age of security", Springer Nature, 2015

[13] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, "Handbook of applied cryptography", CRC press, 2018

[14] L. Lehtomäki, "Realizing eID scheme on TPM 2.0 hardware", 2016

[15] "TPM 1.2 vs 2.0: Here's everything you need to know." `https://windowsreport.com/tpm-1-2-vs-2-0/`

[16] Windowsreport, "TPM 1.2 vs 2.0, what are the differences?." `https://windowsreport.com/tpm-1-2-vs-2-0/`

[17] Trusted Computing Group, "Trusted Platform Module (TPM) 2.0: A Brief Introduction." `https://www.trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-A-Brief-Introduction.pdf`

[18] Trusted Computing Group, "TSS 2.0 Overview and Common", TCG Published, October 1, 2021

[19] S. W. Smith, "Outbound authentication for programmable secure coprocessors", European Symposium on Research in Computer Security, 2002, pp. 72–89, DOI 10.1007/3-540-45853-0_5

[20] D. D. Clark and D. R. Wilson, "A comparison of commercial and military computer security policies", 1987 IEEE Symposium on Security and Privacy, 1987, pp. 184–184, DOI 10.1109/sp.1987.10001

[21] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn, "Design and implementation of a tcg-based integrity measurement architecture", USENIX Security symposium, 2004, pp. 223–238

[22] N. Schear, P. T. Cable, T. M. Moyer, B. Richard, and R. Rudd, "Bootstrapping and maintaining trust in the cloud", Proceedings of the 32nd Annual Conference on Computer Security Applications, 2016, pp. 65–77, DOI 10.1145/2991079.2991104

[23] "Keylime Documentation." https://keylime-docs.readthedocs.io/_/downloads/en/latest/pdf/

[24] "Keylime Project Description." https://pypi.org/project/keylime/

[25] "TPM2-tool Manual." https://github.com/tpm2-software/tpm2-tools/tree/master/man

[26] S. Varga, "Establishment of cryptographic identities in cloud via keylime tool", 2019

[27] "Raspberry Pi Documentation: The Linux Kernel." https://www.raspberrypi.com/documentation/computers/linux_kernel.html#using-menuconfig

[28] "TPM 2.0 in U-Boot on Raspberry Pi 4." https://github.com/joholl/rpi4-uboot-tpm

[29] "OPTIGA TPM Application Note: Remote Attestation." https://github.com/Infineon/remote-attestation-optiga-tpm/blob/master/documents/tpm-appnote-ra.pdf

[30] "Forecast end-user spending on IoT solutions worldwide from 2017 to 2025." https://www.statista.com/statistics/976313/global-iot-market-size/