POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Continuous Remote Attestation Process in the scenario of the ROOT time precision distribution network

Supervisors
Prof. Antonio Lioy
Dr. Ing. Diana Gratiela Berbecaru

Candidate
Silvio GIROLAMI

MARZO 2022

To my parents, for all their sacrifices and for believing in me even when I didn't have the strength

Abstract

Nowadays the trustworthiness, accuracy and availability of time information distributed over networks of every kind is becoming more and more relevant due to the birth of several applications that require accurate and secure time and an high-precision synchronization among their components. Several kinds of threats can be identified in this type of distribution networks, attacks can be run starting from the transmission of the time data until the moment when the information is elaborated by the node.

A particular type of networks in which time becomes of primary importance are the 5G telecommunications networks. In order to provide the expected Quality of Service, synchronization is needed in time-precision networks like this. Together with the distribution issue, the elaboration of the information received, that has to be forwarded right after, become crucial too. Some telecommunication networks are classified as critical infrastructure thus need particular robustness against cyberattacks.

In this paper, we identify several type of attacks that can be run against these networks focusing our attention on software attacks that are run trying to take control of a particular machine or simply to install some code on it or to change its configuration. The goal of the thesis is to find a way to detect if an attack like this happens and to report this as quickly as possible, by monitoring the platform status constantly and notifying an anomaly as soon as it is detected, following the principles of the Remote Attestation process.

Summary

More and more systems such as broadcasting networks, power grids, financial and banking applications, transport networks and also telecommunication networks, are gradually increasing the demand for **accurate time synchronization** and time distribution solutions, even if with different requirements in terms of accuracy and precision [2]. Probably one of the systems, among those listed above, with the more restrictive requirements are the telecommunication networks since, with the advent of the 4G technology, the need for accuracy in the distribution of time has grown. This growth became even more significant with the latest technology in this field: **5G**, which requires the presence of multiple timing sources in the whole network. In a scenario like this, great importance is given to the way in which time is distributed across the network and how synchronization among nodes is obtained, and most important whether they meet the fixed requirements or not.

It is precisely these requirements that are the subject of the study of the **ROOT** project, whose goal is to analyse the impact and effects of cyberattacks against transport-based distribution architecture [1].The architecture under consideration is based on satellite-derived timing information which is received by Global Navigation Satellite System(GNSS) that can distribute the absolute time reference, through dedicated precision transport protocols like *PTP or WB-PTP* (an enhanced version of PTP), to terrestrial really accurate celsium clocks installed on each node of the network[5].

Since time information plays such an important role in the scenario of 5G telecommunication networks, it represents a security asset to be protected against every type of attack that could undermine time distribution [22]. Among all the possible kind of attacks, hardware-based, network-based and time protocol-based attacks, the ones on which this thesis focuses on are the *software tampering attacks*: these kind of attacks aim to gain access into the targeted machine, usually a node of the distribution network, and modify configuration files or executable stored in the machine or to download malicious code.

The goal of ROOT is not necessarily to prevent these attacks from happening but to detect when they happen and identify the point of failure in the network in order to limit the damages by isolating that infected node: to do so, the *remote attestation process* is used. It consist in an administrator willing to attest the trusted status of a remote machine, verifying that the software running on it and its configuration files are the expected ones. A fundamental component of this process is the **Trusted Platform Module (TPM)** that is an inexpensive chip that can perform cryptographic operations. The reference software for Remote

Attestation is **Keylime**, which is an open source framework based on roles like the Verifier, the Tenant and the Registrar.

The objective of the thesis is to create a testbed composed by several nodes, each of which has a modified version of Keylime running on it, and to carry out the remote attestation process simultaneously on every node created. In doing so the goal is to verify that, despite the delay introduced by the RA process, the requirements of precision and accuracy of the time information for a 5G telecommunication network are met.

Contents

1 Introduction 7

1.1 Problem statement 7

1.2 Objectives 8

2 Trusted Computing 9

2.1 What is Trusted Computing? 9

2.2 Trusted Platform Module 10

2.2.1 TPM 2.0 - Architecture 11

2.2.2 TPM 2.0 - TSS 14

2.2.3 TPM 2.0 - Roots of Trust 16

2.2.4 TPM 2.0 - Keys Hierarchies 18

3 Remote Attestation 20

3.1 Integrity Measurements 20

3.2 Remote attestation 25

3.2.1 Activation of credential 26

3.2.2 Attestation process 28

4 The ROOT Project 31

4.1 GNSS-based time synchronization solution 31

4.1.1 The GNSS-based solution architecture 32

4.2 Attacks against GNSS-based architectures 34

4.2.1 Classification of attacks against GNSS RF Spectrum 34

4.2.2 Classification of attacks against the Distribution Network . . 35

5 The Keylime Framework 39

5.1 Background 39

5.2 Design 41

5.3 Keylime nodes' architecture 41

5.4 Keylime and the ROOT project 48

6 Keylime Continuous Remote Attestation of a physical node with a physical TPM 50

6.1 Prototype 50

6.1.1 Prototype bootstrap process with u-boot 51

6.2 Architecture of the proposed solution 54

6.3 Modifications made on the Keylime Framework 56

7 Keylime Continuous Remote Attestation of a physical node with a TPM Simulator 59

7.1 Devices involved in the analysis 59

7.1.1 TPM2.0 Emulator and absence of the CRTM 60

7.1.2 TABRMD : Access Broker and Resource Manager Daemon . 61

7.1.3 IMA and TPM2.0 Emulator 64

7.2 Architecture of the solution 65

7.3 Modifications made on the Keylime framework 67

8 Keylime Continuous Remote Attestation of the ROOT infrastructure 70

8.1 Description of the proposed Architecture 70

8.2 Modifications made to the Keylime Framework 72

8.3 Attacks to the testbed and response from the framework 75

8.4 Tests and performances 77

8.4.1 Performance evaluation 79

9 Conclusions and future works 83

Bibliography 85

A Prototype characteristics 88

B Keylime installation on a platform with a physical TPM installed 90

B.1 Keylime Trust Agent Installation 90

B.1.1 Tpm2-tss 91

B.1.2 Tpm2-abrmd 92

B.1.3 Tpm2-tools 93

B.1.4 Installation of the Keylime Agent 94

B.2 Installation of the Keylime Tenant, Keylime Verifier and Keylime Registrar framework on one single node 95

B.3 Launching Keylime and starting the attestation process 97

C Keylime installation on a platform without a physical TPM and the abrmd installed 99

C.1 Keylime Trust Agent Installation . . . 99

C.1.1 Tpm2-tss . . . 100

C.1.2 Tpm2-tools . . . 101

C.1.3 TPM 2.0 Emulator . . . 101

C.1.4 Enabling IMA . . . 103

C.1.5 Installation of the Keylime Agent . . . 103

C.2 Installation of the Keylime Tenant, Keylime Verifier and Keylime Registrar framework on one single node . . . 105

C.3 Launching Keylime and starting the attestation process . . . 107

Chapter 1

Introduction

1.1 Problem statement

The birth of new technologies in the field of telecommunications networks, like 5G, brought to life lots of projects trying to exploit those novelties. In this particular context the ROOT project came out, trying to combine both the new 5G network and the benefits brought by the Galileo GNSS system. The idea of this project is to obtain a better distribution of the time through the network, in terms of accuracy, trying to build an high-precision synchronization based on Galileo, and in particular on its authenticated signals.

The most relevant aspect that 5G introduced with respect to timing and synchronization is that it opens to the possibility of building telecom networks that are much more resilient, bringing an increased level of robustness to the GNSS-based timing sources [1]. This purpose is achieved passing through 4 pillars:

- *5G telecom networks*: they introduced more stringent requirements to time and phase synchronization. These requirements are satisfied by the use of Distributed Granmaster Clock(D-CMC);

- *Reliable GNSS receivers*: new type of receivers are employed in the building of the network providing much more security, in particular against jamming and spoofing attacks, being able to work at different frequencies;

- *New PTP mechanisms*: a new protocol for a more accurate long-distribution of time is used, **WR-PTP**. This protocol has an accuracy of nanoseconds and a picoseconds precision of synchronization by extending the PTP protocol [21];

- *Network security*: in order to verify the trustworthiness of the nodes of the network, integrity check and remote attestation mechanisms are introduced relying on trusted computing. Time becomes an important security asset and, sure enough accordingly to Guy Buesnel, a PNT security technologist:

  > *Poor understanding of network timing can create big risks for organisations, especially if they are managing critical infrastructures that nations rely on.*

The work underlying this thesis is all about the fourth pillar and in particular is focused on the remote attestation process to be done over the GNSS receivers and the related management nodes.

The ROOT project aims to bring security in the whole network, starting from the moment in which the time information is generated until the moment in which it is received and elaborated, so working in a *dividi et impera scenario*, just focusing on the computation made on a single receiver, it is possible to assume that the time-related data that reaches the receiver is the original one, which means that it wasn't tampered with from the satellite to the device. Assuming this, what has to be done is to verify that the node, which is going to make some computation based on the time, behaves as expected and that the software running on it and its configuration are correct. This process is called *Remote Attestation* and can cope with attacks targeting the software but is ineffective with attacks aiming the hardware. The framework that will be used in order to perform this remote attestation process is *Keylime*: an open source framework thought for distributed infrastructure.

In the next chapters the concepts of *Trusted Computing*, *Trusted Platform Module* and *Remote Attestation* will be introduced, since they represent fundamentals concepts to better understand how the reliability of a device can be assessed.

1.2 Objectives

The main objective of this thesis is to illustrate the functioning of the *Keylime framework*, trying to explain the reasons why it was chosen to implement the attestation process in this particular scenario, outlining its strengths and the features that makes it perfectly suited for the context of the ROOT project.

Once the framework will be described, it will be possible to understand that ROOT's specific architecture, and in particular the devices that compose it, require some modifications to the code in order to have it working properly in this scenario. The main problem will be identified in the *low performance devices* that make up the architecture: these are simple devices that very often have neither a physical TPM nor the support for the installation of a *resource manager* or an access broker. This lack of resources bring us to the modification of the framework: the goal of this modifications is to overcome both the absence of a way to manage the small memory available inside the TPM and the absence of communication between the TPM Emulator and IMA.

Once the necessary modifications have been defined to make up for these shortcomings, the final objective is to ensure that the framework, running a single daemon, can simultaneously certify both devices with a physical TPM on board and devices with a TPM emulator. This is very important because the ROOT project architecture contains both the kinds of device and it is necessary that they are attested by the same process, without introducing excessive performance penalties for Keylime.

Chapter 2

Trusted Computing

2.1 What is Trusted Computing?

The concept of Trusted Computing (TC) includes several cybersecurity key ideas such as *integrity, authenticity and confidentiality*; in a more generic way it is linked to the desire of being able to determine whether a system can be considered trusted of not. It first came out in the middle of 80's, when the concepts of *Trusted Computing Base and Security Parameter* were presented for the first time in a paper published by the Department of Defense (DoD) [3]: these ideas were ahead of the time, since the security of the years was mainly a physical security. When Internet began to spread in the 1990s and became more easily accessible, those ideas started to be a matter of concern and so the big players in the market ran for cover with software solutions to protect PCs. Of course these kind of solutions were not enough: software solution by their own were not sufficient to guarantee the trusted state of a Personal Computer since they are to easy to compromise. Hardware solutions were needed
This conviction was the starting point for the work of the *Trusted Computing Group* (**TCG**) since it was born in the first years of the new century. TCG is an organization that promotes the development of tools and standards for the trusted computing technology and defines the guidelines for the concept of Trusted Platform, which heavily depends on the *Trusted Platform Module* (**TPM**), an extremely important chip presented by the TCG in the August of 2000 and now widely used. This chip is the key point of the Trusted Computing System since it conveys important mechanisms and features, like the secure storage, the chain of trust, cryptographic operations, secure authorization and roots of trust. Since it is an hardware component, it is not vulnerable to the same kind of breaches that a software component is and, most importantly, can assess the status of the operating system at bootstrapping time. Over the years, several versions of the TPM have been released, each one trying to improve the previous with new features: in 2002 version 1.1b came out, in 2004 version 1.2 was released and was a huge success and within a few years was installed on practically every computer produced. Starting from 2005, with the advent of the first attacks against SHA-1, the TCG began to implement version 2.0, since the previous version was heavily based on SHA-1.

Following the Trusted Computing Group definition, a Platform can be considered *trusted* if its behaviour is the expected one, of course this is a necessary but not sufficient condition for trustworthiness. It's also necessary to determine the identity of a platform, taking into account its software and hardware components, that should behave in a predictable way.

Going deeper in the concept of trusted platform, the TCG defines some principles a system should be compliant with to be considered trusted, and these principles are [7]:

- *Secure Input and Output*: All the information that are exchanged through the bus, since it is shared, should be ciphered.
- *Memory Curtain*: There are some kind of information, like private or symmetric keys, that should stay private and not accessible even by the Operative System. So there should be a kind of strategy, whose implementation depends on the vendor, to keep these information isolated in the memory and not accessible if not authorized.
- *Endorsement Key*: It is an asymmetric cryptographic key pair which is stored in the TPM by the manufacturer at creation time. It never leaves the TPM and cannot be accessed externally but can be only used in signature and encryption operations.
- *Sealed Storage*: This is a particular type of encryption which is not only based on the encryption key but also on the status of the system, both software and hardware. In this way is it possible to decrypt a portion of the memory only if you have the key and the system is in the same status that it had when encryption was made.
- *Remote Attestation*: It is a process whose goal is to verify remotely that an authenticated system is not compromised, that it was not tampered with and that its behaviour is the one expected.
- *Trusted Third Party(TTP)*: This is a key concept in the TCG specifications since a TTP allows a verifier to access the identity of a platform without exposing any private information of the TPM or of the platform hosting it.

2.2 Trusted Platform Module

Created in the 1990s by a group of engineers which became later the starting point of the TCG, the Trusted Platform Module is a cryptographic coprocessor that is now present in almost all devices. The idea behind this chip was to answer to the great spread of personal computers and electronic devices of those years, in particular because until that moment security had never been an issue: TPM was the way to protect device's cryptographic assets [10]. Among all the features it has, the most relevant one, for the remote attestation process, is that the TPM must be secure itself, in this way it can be seen as a *Root Of Trust* being able to provide confidentiality, protection and integrity even if it's not robust against hardware-based attacks. Alongside with this feature, the other key functionality of the TPM is *identification*, which means it is able to manage identity keys, in particular its endorsement key pair that represent its own identity: this is crucial in the remote attestation process since each TPM is responsible for report its own state to the verifier, and this is done with a signature made with the EK.

There are two major versions of the TPM itself, the 1.2 and the 2.0 (from 2003 and 2014) where the 2.0 is the evolution of the 1.2 and alongside with some new features, it introduces much more cryptographic algorithms, especially because of the weaknesses of SHA-1. For these reasons the TPM 2.0 and its characteristics will be presented in the following.

2.2.1 TPM 2.0 - Architecture

Probably the best way to understand the capabilities of the TPM is to study its structure in order to understand what parts it is made of and what each part does.

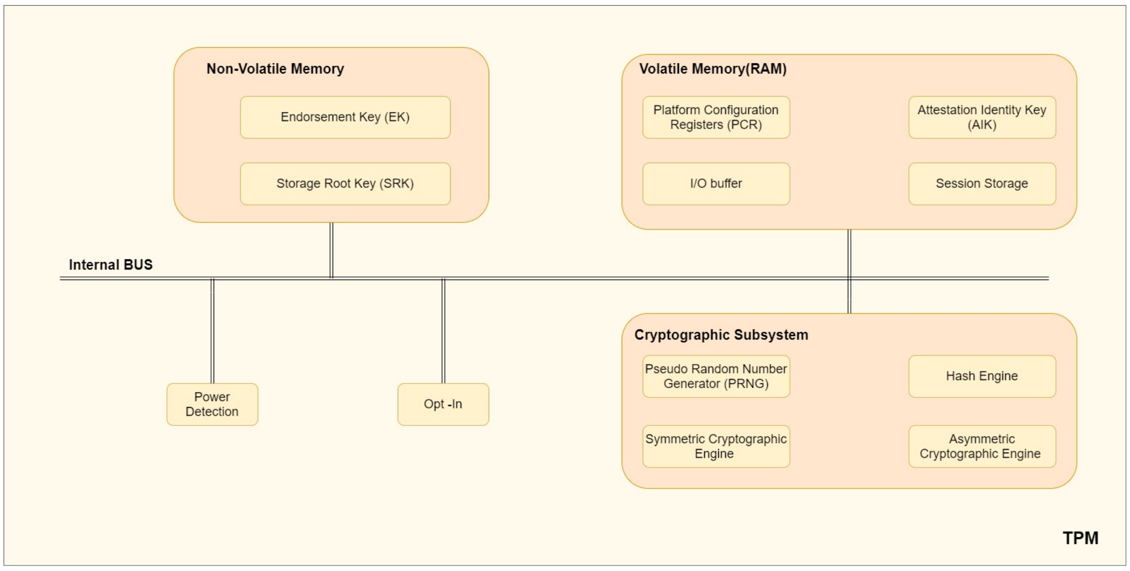


Figure 2.1. TPM 2.0 Internal architecture (source [11] [12])

TCG specifications report that it has several components and they are all connected by an **internal bus**. The communication between the system and the TPM itself is made through the **I/O Buffer**: here the system places commands data for the TPM and the chip puts responses data for the external components. All commands that are inserted in the buffer, which have a well-defined structure, are validated and an authorization check is made on the memory location that has to be accessed. This last check is done by the **OPT-IN** component, which maintains flags to manage access rights and TPM routine.

Probably the most important components for this thesis are the **Platform Configuration Registers (PCR)** which are shielded locations of memory used to validate the measurements made by the system to its own internal components. These registers are used in the field of remote attestation since they contain hashes which are measurements of software components of the whole system. The number of PCR is usually 23/24, but is actually variable depending on the vendor specifications, and each of them is used to store the hash of a specific part of the system, as specified in the *table 2.1* [6]

PCR Number	*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 2.1. List of PCRs and the corresponding content

As already said, the content of these registers is the measurement of some piece of software, regular file or configuration files, done through an hash; the peculiar thing about PCRs is that their content cannot be modified if not at reboot time or with a particular operation called *extension*. The value is cleared every time the TPM is restarted, usually the value is set to all zeros, and it cannot be erased of modified manually, extension is the only operation that can be done. This is a one-way operation that involves the hash function

PCR new value = Hash(PCR old value || data to extend)

this means that, given the prc new value, is not possible to go back to its old value. The content of PCRs can be simply read internally or externally to validate the state of the system, usually through an operation of quote which is part of the process of the remote attestation. PCRs can be collected in banks, each bank of PCRs is extended with the same Hash algorithm so a bank is identified by the Hash algorithm it uses.

As pointed before, one of the fundamental feature of the TPM is to manage identity and to do so it needs to store cryptographic keys: they are stored in the **Non-Volatile Storage**. This memory keeps two long terms keys, the **Endorsement Key(EK)** and the **Storage Root Key(SRK)**: the first one is used in some critical tasks like the remote attestation process, the second one conveys the secure storage into the TPM and is strongly linked to the TMP users in the sense that changes every time the *Take Ownership* procedure is executed (a new user takes the ownership of the TPM). Some NV memory can be also available for allocation and to store owner authorisation data, like the owner-password.

The TPM also contains a **Volatile Memory** that holds transient data, which can be of various type:

- A part of the TPM RAM is dedicated to the I/O buffer;
- PCRs are stored here;
- Object store: TPM RAM also contains data and cryptographic keys that can be passed as parameters to commands like *TPM2Load()* or *TPM2CreatePrimary()*;
- Session Store: TPM create some sessions to build a queue of operations. Each session has some data associated with such as keys or some encrypted data.

A **Power Detection Module** manages the TPM power states (the only two available states are ON and OFF). Following the TCG specification, the TPM should be notified whenever the power states of the platform changes.

The last component, or better group of components, is the **Cryptography Subsystem**, that implements all TPM's cryptographic functions. It is possible to identify 4 major components of this group:

- **Hash Engine**: this is probably the most relevant one for the remote attestation process since it s the one that implements the expand function. In any case, hash functions may be used also in other operations, even by external entities. Specification by the TCG requires SHA-1 and SHA-256 to be available.
- **Asymmetric cryptographic engine**: a TPM must be able to perform RSA cryptographic operations such as the digital signature, since they are necessary for identification during the reporting phase of the RA process, but also encryption, decryption and obviously verification are supported. To be compliant with the specification, TPM has to support RSA-2048, ECC using the Barreto-Naehrig 256-bit curve and the NIST P-256 curve.
- **Symmetric encryption engine**: Symmetric operation are also used by the TPM that could need to encrypt some command parameters. The only mandatory block cipher mode according to specifications is CFB while the required algorithm is AES with a key of at least 128 bits.
- **Random Number Generator**: it is the source of randomness of the chip, used for gathering entropy to create keys or nonces. It actually is a Pseudo RNG (PRNG).

Now that the internal structure of the TPM has been described, it is supposed to be easier to understand the security capabilities it has and the problems it tries to address [10]:

- **Identification of devices** Thanks to the Endorsement Key and to the RTR, the TPM conveys a new and more secure method, that differs from the MAC of IP addresses, to identify a node in a network;

- **Secure Generation of Keys** Due to the presence of the Random Number Generator inside it, the TPM can also provide a way to generate keys securely;
- **Secure storage of keys** As we saw, the TPM contains some shielded locations of memory, it is possible to store inside it several object. Since these shielded locations are accessible only through Protected Capabilities, data stored within are protected from unauthorized deletion, modification or disclosure;
- **NVRAM storage** The presence of a Non-Volatile memory allows the TPM keep a copy of the EK certificate without the risk of loosing it;
- **Integrity platform attestation** Probably the most relevant feature for this thesis scope, the TPM conveys an hardware-based solution to attest the integrity of a platform, avoiding the classical problems of a software-based solutions.

2.2.2 TPM 2.0 - TSS

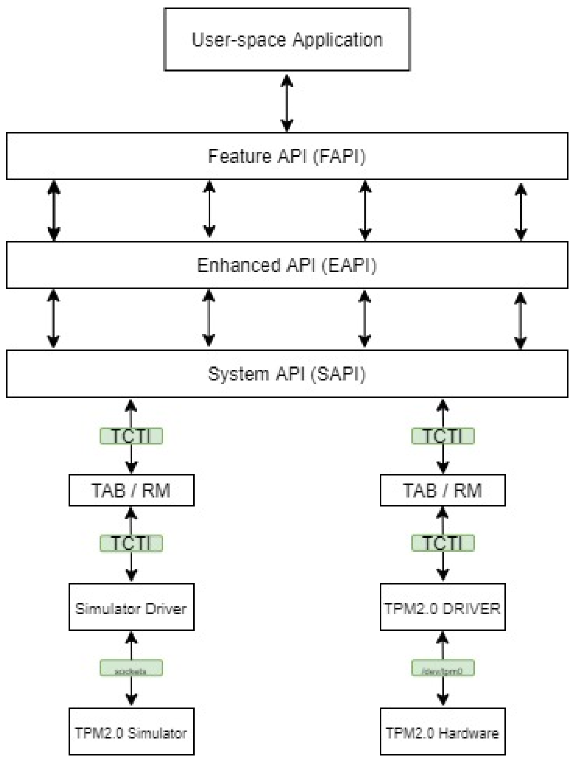


Figure 2.2. TSS structure (source [13])

In order to make the interaction with the TPM more usable and intuitive, the TCG came up with the implementation of the **TCG Software Stack** (TSS) which provide some standard APIs for accessing the functions of the TPM easily. In this way developers can use this software specifications to create client applications calling the core functionalities of the TPM, like the primitives for key generation or the collection of the measurements, in a trusted environment without knowing low level details of teh TPM. The main goal is to provide a standard set of APIs for Application vendors to use TPM which can be unrelated to the underlying hardware: by writing specifications considering an abstraction of the hardware differences, is possible to have APIs working regardless of the hardware, OS or environment. TSS is composed by several layers to be scalable both for high performance end devices and low end systems.

Every layer of the stack has a different level of abstraction, the closer to the application the higher the level of abstraction:

- *Feature API (FAPI)*: the first layer after the application, the one with the higher level of abstraction which provide the 80% of the APIs needed by the programmer. Since the high level of abstraction, the number of parameters needed and the number of calls to be done are reduced: this provide a way of programming using the TPM that is as easy as possible.

- *Enhanced System API (ESAPI)*: this is an intermediate level whose goal is to reduce the complexity required to call the system level APIs from the next layer (the SAPI), but still allowing the call to cryptographic operations(like the HMAC, encryption or decryption operations) [14]. Even if the ESAPI is much easier to use than the SAPI, it still requires a in-depth understanding about the TPM2.0 interface.

- *System API (SAPI)*: this layer provide a complete mapping of the TPM2 commands, making available to the programmer the access to the whole set of functionalities of the cryptographic coprocessor. It implements the remaining 20% of the APIs with particular attention to the low level calls, but a deep knowledge of the TPM structures is required. Both synchronous and asynchronous API are exposed.

- *Marshalling/Unmarshalling (MUAPI)*: This API provides a set of marshalling and unmarshalling functions for all data types, operations that are needed both by the SAPI and the ESAPI, that's way it has it's own layer.

- *TPM Command Transmisison Interface (TCTI)*: This is a linking layer between the hugher and the lower level of the stack, it doesn't expose any API but handles all the communication to and from the lower layers of the stack, providing a standard interface to transmit/receive TPM command/responses.

- *TPM Access Broker (TAB)*: this is the layer that manages the accesses to the TPM handling multi-process synchronization, serving one process at a time, with no interferences from others.

- *Resource Manager*: since more than one process can interact with the TPM at the same time, a layer that can act as a virtual memory manager is needed. The TPM has a really limited amount of memory, so the resource manager is the component that swaps in and out different sessions and objects related to different processes when the TPM passes from serving the request by one process to serve the request from another one.

- *Device Driver*: it is the layer directly connected to the TPM and is the OS-specific driver that handles all the direct communication with the device, like the handshake. It sends requests to the TPM and receive responses to be sent to higher level.

2.2.3 TPM 2.0 - Roots of Trust

As already stated, one of the main ideas behind the creation of the TPM was the desire of being able to create a Chain of Trust starting from a Root of Trust, a component that can be considered trusted since its misbehaviour cannot be detected: the TPM is the solution to this. A Root of Trust is the minimum set of system components that conveys trustworthiness to a platform and its strongly bounded to the concept of *Trusted Building Block*.
Since the characteristics that affects the trustworthiness of a platform are multiple, it would be more appropriate to talk about *roots of trust* where each one of them is the starting point for one of the parts in which the process of trust attestation is divided. According to its specifications, the TCG group considers a platform trusted if it has three roots of trust, which are:

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

The only thing that can be verified about a root of trust, since it is trusted by default, is its implementation that can be properly verified and certified as trusted.

RTM

The first phase of the *attestation process*, whose goal is to define a platform as trusted, is the **integrity measurements**, which is the concept behind the construction of the chain of trust: every component measures the next one (*Transitive Trust*). The role of the RTM is to make the needed measurements and to update the PCRs through the extension operation: in this way information are passed to the RTS. The problem here is to define the first ring of the chain of trust: the TCG identifies the Core Root of Trust for Measurement (CRTM). The RTM is actually the computing engine of the platform (generally the CPU) and it is controlled by the CRTM, which is outside the TPM, usually in the BIOS or directly in the motherboard, and contains an immutable piece of code (the RTM code): it is the first set of instructions executed when a new chain of trust is created and its goal is to check the system environment at boot time in order to attest the initial trust state of the system [4].

- *Static-RTM*: every time we want to initialize a root of trust, a reboot of the system is needed. When the system is rebooted, once the boot phase is completed, it ends in a known initial state whose measurement can be checked and validated easily, starting a chain of trust. It is called Static since the first ring of the chain is the state at reboot time, which is immutable.

- *Dynamic-RTM*: the dynamic means that it is not necessary that the system is rebooted to start a new chain of trust since the CPU can act as the CRTM and protect some new portions of the memory, after they are measured, creating a new RTM dynamically. The major advantage of the dynamic solution is the fact that chains are much shorter, since there is no need to reboot the system any time, and that it is possible to have intermediate trusted state to start measurements from.

RTS

The second step of the attestation process is to collect the measurements done on some object and store it into the PCR by performing the extend operation. The idea is to have a *shielded location of memory* inside the TPM, which is accessible only by the TPM itself and is protected by the SRK owner-bound asymmetric key pair, that keeps integrity measurements and protects data and cryptographic keys.

RTR

The last phase of the attestation process is the reporting one. There are two crucial points in reporting: sending the measurements taken properly and in a secure way and being able to bind the measurements sent to TPM identity. The RTR is responsible for this. This process represents the interaction between the RTR and the RTS: the former has to report the content of the latter and this is usually done by sending the digitally signed content of the needed PCRs (through a *TPM2Quote()* for example). The digitally signature is the means by which we have the binding between the measurements and the TPM identity and it is done with the *Attestation Identity Key*(AIK), derived by the EK [12].

There are some key concepts around the idea of Trust which are particularly important for this discussion:

- *Trusted building Block (TBB)*: the CRTM and the TPM together make up the TBB which is required to instantiate a Root of Trust;
- *Trusted computing Base (TCB)*: is the set of components (HW and SW) that manage the compliance to the security policies. Its peculiarity is that it cannot be tampered with by an external component and the TPM can be configured to not start if the TBB is not initialized in the correct way.
- *Trust Boundaries*: the TBB together with the Root of Trust build the trust boundary. It can be extended, if needed, when the CRTM needs to use another element and before using it, it should be trusted: to have this a measurement operation is done and if it is successful the trust boundary is expanded.
- *Transitive Trust*: this is the operation underlying the attestation process and integrity measurement. A trusted component, that is the root of trust in the first step, can measure another component and once it has verified that

the measurement is the expected one, transfers its trustworthiness to that component, that in turn can measure another component, creating a chain of trust and expanding the trust boundary.

2.2.4 TPM 2.0 - Keys Hierarchies

The word hierarchy refers to a set of objects that are managed as a group. With a view to trusted platform, a key hierarchy is a collection of objects like a seed, primary keys and proof values. The seed is probably the most important element of the hierarchy since it is its starting point: it is a large random number that never leaves the TPM from which primary keys are generated. From these primary key other objects are derived, the proof values, which are used by the TPM to attest its identity when it sends some data. There are three types of persistent hierarchies (they have a persistent seed and primary keys remain the same unless a reboot of the TPM is performed) that are supported by the TPM:

- **The platform hierarchy**: used to ensure the integrity of the system firmware;
- **The storage hierarchy**: used by the platform owner for a variety of non-privacy-sensitive purposes;
- **The endorsement hierarchy**: used when the user wants to ensure the integrity of privacy-sensitive data, attesting TPM identity since its primary keys are guaranteed to be constrained and unique to each TPM by the manufacturer. The keys of this hierarchy are also known as *Endorsement Keys* (EKs) and are created starting from the unique *endorsement seed* contained in the TPM. These keys are also used in critical tasks as the remote attestation process.

The EK is never erased, never leaves the TPM and in order to be more protected is used to generate the Attestation Identity Key (AIK), that is a kind of ephemeral key pair that can be created many times to prevent traceability, that is the actual key used to digitally sign the information sent by the RTR (through the Quote operation). Usually, to attest the actual identity of the platform, together with the digitally signed digest, an Endorsement Key certificate is sent along with the EK certificate.
The Endorsement Hierarchy and the EK have a key role in the identification of a platform: the RTR, and the reporting phase itself, are meaningless if not strictly linked to the TPM they refer to. In this sense it becomes really important to find a way to securely identify a TPM (which means a RTR) and to link the RTR to the RTM that performed some measurements. This is all about the need for *Secure Identity*, that is a proof of the physical bounding between the RTM and the RTR and is needed to be sure that a particular quote was sent by a well defined TPM. This is achieved through the EK, since it is almost impossible to have two TPM with the same endorsement seed (that means same endorsement keys), and also to the EK certificate, released by the TPM manufacturer and stored inside the TPM itself.

There is another type of hierarchy that is supported by the TPM, the volatile *null hierarchy* whose seed is regenerated, together with the primary keys, every time the system is rebooted; it is used when the TPM is employed only as a cryptographic coprocessor and it accessible by anyone.

Chapter 3

Remote Attestation

3.1 Integrity Measurements

Starting from the already described concepts of *Transitive Trust* and *Trust Boundary* 2.2.3, we can define *Integrity measurement* as the operation that aims to create transitive trust through the various components of the platform. The security pillar on which it is based is *integrity* of course: the objective is to be sure that the file (executable or not) that is about to be opened is the expected one and it was not tampered with.
We have already identified the CRTM as the root of the chain of trust, for the same reason it is also considered as the first step of the integrity measurement process, since it has the duty to start the bootloader and the BIOS. Once the CRTM is executed, every object that is called next, is measured just before being accessed and has the duty to measure the next one to be opened, following the transitive trust concept. Every measurement has to be recorded into a PCR, performing the already mentioned *extend* operation: it is performed onto the designed PCR, whose new value will be the hash of the concatenation of its old value and the measurement done on the just measured object. In this sense, the extend function is a one way operation and cannot be undone, keeping track of all the accessed files, since the boot of the system, inside a PCR.
So the three phases the process of Integrity Measurements is divided in are:

1. The CRTM is the trusted component by definition, the only one that is not measured and therefore is the first executed. It is the one that measures (i.e. computing the digest) the rest of the BIOS and of the bootloader, than the control is passed to the latter [12];

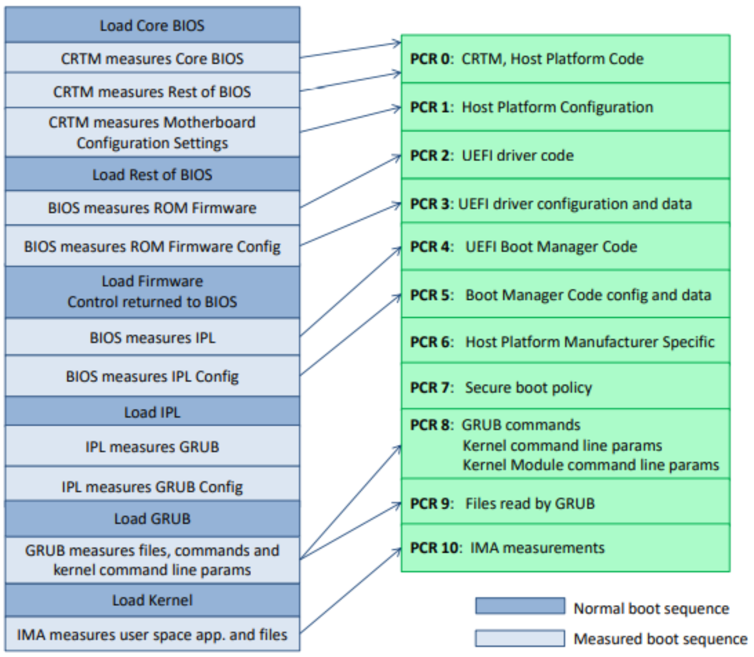
2. The boot loader computes the digest of the OS kernel and any additional code that is needed, and just before executing it the proper PCR is extended with the digest evaluated;

3. Once run, the OS kernel measures whatever application, executable, configuration file and any other data that is accessed by the user and, once the measurement is extended into the PCR, the process goes on with the next object.

These three steps can be divided, following a temporal logic, into 2 groups: the first two are related to what happens before the OS starts, while the last one is about the operations performed by the OS. The first group is the so called *Measured Boot*, while the second one is handled by the *Integrity Measurements Architecture (IMA)*.

Measured Boot

The *Measured Boot* is the first part of the integrity measurement process. It consists in the measurement of all the files (both executable and configuration ones) that are involved in the bootstrapping phase of the system: all these files are measured and the resulting digest is stored in a specific PCR (following the guidelines contained in Table 2.1). The order in which files are accessed and software components are called in this phase is static, which means that the order is always the same, as well as the PCR where each measurement is stored (Figure 3.1):



Load Core BIOS
CRTM measures Core BIOS
CRTM measures Rest of BIOS
CRTM measures Motherboard Configuration Settings
Load Rest of BIOS
BIOS measures ROM Firmware
BIOS measures ROM Firmware Config
Load Firmware Control returned to BIOS
BIOS measures IPL
BIOS measures IPL Config
Load IPL
IPL measures GRUB
IPL measures GRUB Config
Load GRUB
GRUB measures files, commands and kernel command line params
Load Kernel
IMA measures user space app. and files
PCR 0: CRTM, Host Platform Code
PCR 1: Host Platform Configuration
PCR 2: UEFI driver code
PCR 3: UEFI driver configuration and data
PCR 4: UEFI Boot Manager Code
PCR 5: Boot Manager Code config and data
PCR 6: Host Platform Manufacturer Specific
PCR 7: Secure boot policy
PCR 8: GRUB commands Kernel command line params Kernel Module command line params
PCR 9: Files read by GRUB
PCR 10: IMA measurements
Normal boot sequence
Measured boot sequence

Figure 3.1. Measured Boot operations order and PCRs involved (Source:[6])

The first component to be called is the only one that cannot be measured and for this reason the *CRTM* was chosen: it measures itself, the BIOS and motherboard configuration settings, extends PCR 0 and 1 and passes the control to the rest of the BIOS.
The *BIOS* has the duty to measure the ROM Firmware and ROM Firmware's configuration files and extends PCRs 2 and 3.
The control is then passed to the *ROM Firmware* and, when the BIOS takes it back, the *Initial Program Loader* (IPL) is measured: usually this is the code of the primary boot loader. The measurement of the IPL is stored in PCR 4 while its data and configuration's measurements are stored in PCR 5.

The control is then passed to the primary boot loader, that calls the secondary one (*GRUB* for x86 platform): GRUB's measurements are stored in PCR 8 together with every operation performed by it, while any file it accesses is measured and extends PCR9. Once the GRUB has completed its operations, the control is passed to the kernel, and the dynamic part starts.

The result at the end of the Measured Boot is to have a chain of trust that goes from the CRTM up to the kernel, but the validation of this chain can be done in two different ways:

- **Secure Boot** : the validation of the measurements is done step by step, which means that every time a PCR is extended the measurements done are checked with the software signatures of the component that is receiving the control of the platform. If the digests match, the control passes from one component to the other, while if the measurement is not valid, the boot process is stopped. In this way the system boots only if it is in a trust state;
- **Trusted Boot**: the boot process goes on as just described and it terminates in any case, since no check is performed until it is completed. When the system starts, its state is properly checked by a third entity which has to verify that all the values contained in the PCRs are the expected one.

Integrity Measurements Architecture

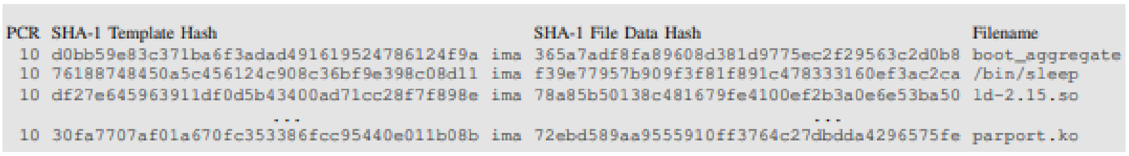
Measured Boot is the solution that covers only the first part of the integrity measurements process, the static part. When the operating system starts, there is a great variety of application and software components that could be accessed in undefined order, that is why the same solution cannot be adopted for the application layer. The transitive trust process is based on the need to know which program, files and in general object is accessed/executed and in which order, but once the operative system is loaded, how do we cope with the dynamic part, the application layer?
Since the whole list of measurements is not available through PCRs that only store a "summary" of all the values, a list of all the digests can be kept; even if it is not mandatory, the TCG strongly recommend it without giving any specification on the implementation. There are several ways in which this could be done, but the most popular TCG-Compliant solutions, and also what this project relays on, is the *Integrity Measurement Architecture (IMA)*, that doesn't require any modification of the Linux operating system.

IMA is an open-source trusted computing component of the kernel integrity subsystem which introduces hooks within the Linux kernel allowing the system to collect hashes of files as soon as they are loaded onto the Linux system, just before being read or executed. Once the hash is evaluated and used to extend the PCR, it is stored into the kernel memory where it cannot be modified by userland applications and here it can be read locally or remotely to verify its integrity [15].

IMA is a really important instrument for the trusted computing process since it allows to extend the static measurements done starting from the CRTM and going on with the BIOS and the Kernel, all the way up to the application level [16] (the dynamic part): it provides a way to know the files accessed, the order in which they were accessed and their measurement. This is possible since IMA maintains a runtime measurement list and thanks to the TPM an aggregate integrity value over this list. This list is called *Measurement Log (ML)* and is a collection of *Measurement Events (MEs)* where each event consists in the measurement of an object. This list is protected by an hash evaluated over it that is stored in a PCRs: an external entity can compare the content of this PCR with an hash evaluated over the list to check that the ML was not tampered with. If list's integrity is verified, the third entity analyses the list entry by entry verifying the integrity of each object.

Enabling IMA is really easy, just add the parameters *ima=on ima_policy=<policy>* to the kernel command line and reboot the system, and the kernel component will run. Once IMA is enabled and running, it won't measure every file, unless specified, but only the files that respects the policies that can be set through the kernel command line parameter *ima_tcb* (if you want the standard ones) or by modifying the *policy* file in the *securityfs* file system in the user space, typically mounted at */sys/kernel/security/ima*. The measure is taken after the object of measurement is opened but just before it is run so it is not possible for it to interfere with the evaluation of the digest.

All the measurements are stored in a list, available both in ASCII and binary, where it is possible to find the path, and the associated digest, of all the component that are measured. These files are called *ascii_runtime_measurement* and *binary_runtime_measurement* and are both available at */sys/kernel/security/ima/*, and their structure is the following one



```
PCR SHA-1 Template Hash                              SHA-1 File Data Hash                     Filename
 10 d0bb59e83c371ba6f3adad491619524786124f9a ima 365a7adf8fa89608d381d9775ec2f29563c2d0b8 boot_aggregate
 10 76188748450a5c456124c908c36bf9e398c08d11 ima f39e77957b909f3f81f891c478333160ef3ac2ca /bin/sleep
 10 df27e645963911df0d5b43400ad71cc28f7f898e ima 78a85b50138c481679fe4100ef2b3a0e6e53ba50 ld-2.15.so
                        ...                                          ...
 10 30fa7707af01a670fc353386fcc95440e011b08b ima 72ebd589aa9555910ff3764c27dbdda4296575fe parport.ko
```

Figure 3.2. ascii_runtime_measurement file structure

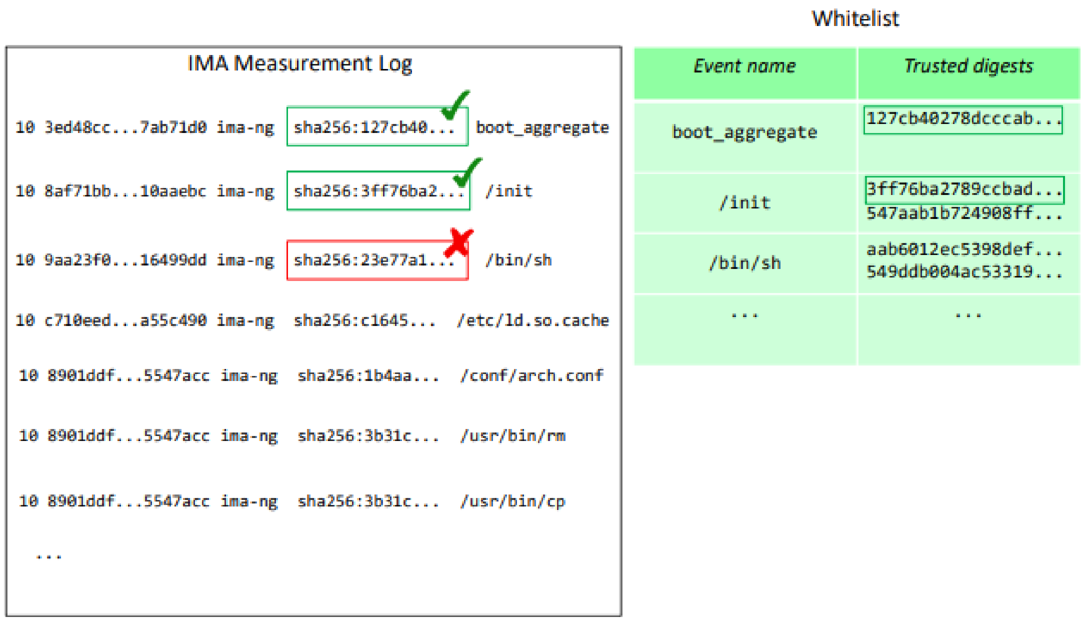
where each column has a particular meaning:

- The first column from left to right is the PCR in which the digest is stored. IMA maintains an aggregate of all the hashes evaluated and usually the PCR in which this value is saved is the number 10 (like in this example);
- The second one is the template hash of the entry, which is a hash that combines the length and values of the file content hash and the pathname.
- The third column represent the template that registered the integrity value (ima in this case).
- The hash generated from the content of the file measured is contained in the last but one column;

- The last column is the file path name of the file measured.

  The first entry of this file is always the *boot_aggregate* (the static part of the attestation process): it is evaluated over the content of the PCRs from 0 to 7. If this aggregate cannot be evaluated (e.g. the TPM is missing) its value is set to all-zeros. This aggregate value is calculated by reading the values inside the PCRs from the bank whose hash algorithm is the one specified in the kernel command line parameter *ima_hash*. The default hash algorithm is SHA-1 and can be changed to SHA256 by booting with ima_hash=sha256.

IMA's ML is generally used by an external entity (*the verifier*) that wants to verify the integrity of the system and its components: to do so the *Validation Mechanism* of the ML has to be performed Figure 3.3. It starts with the check of the hash evaluated over the whole list: it the value is correct, the verifier can be sure that the ML was not tempered with and the process can continue. The verifier keeps a *whitelist* containing all the trusted values (hashes) of the files that can be accessed and measured, so when the validation moment comes, he starts checking entry by entry: if the path inside the ME is contained into the whitelist, and the corresponding measurement is also contained into that list, the validation is successful. If the path is not found, it means that an untrusted file (it may be new or it may never have been opened before) was accessed and the validation fails. While if the hashes don't match, it means that the file has been modified, by its owner or by an attacker.



```
10 3ed48cc...7ab71d0 ima-ng sha256:127cb40... boot_aggregate
10 8af71bb...10aaebc ima-ng sha256:3ff76ba2... /init
10 9aa23f0...16499dd ima-ng sha256:23e77a1... /bin/sh
10 c710eed...a55c490 ima-ng sha256:c1645...  /etc/ld.so.cache
10 8901ddf...5547acc ima-ng sha256:1b4aa...  /conf/arch.conf
10 8901ddf...5547acc ima-ng sha256:3b31c...  /usr/bin/rm
10 8901ddf...5547acc ima-ng sha256:3b31c...  /usr/bin/cp
...
```

Event name	*Trusted digests*
boot_aggregate	127cb40278dcccab...
/init	3ff76ba2789ccbad... 547aab1b724908ff...
/bin/sh	aab6012ec5398def... 549ddb004ac53319...
...	...

Figure 3.3. Validation Mechanism for the Measurement List

3.2 Remote attestation

When you find yourself in a cybersecurity environment, it is very common to need to be sure that a system can be considered trusted, that all the software components that are running or the ones already run were not modified, that its status is the expected one and its configuration is correct. But the concept of attestation gains much more sense and importance when there is an external entity, like a network administrator, who wants to be sure that a particular node of a network is secure, and not only the platform itself: in this case the process of attestation is called **Remote Attestation**.

There are several kind of *attestations* but the most used and also a really simple one is the *Attestation of the platform* which is done using the content of the TPM's PCRs. It is a simple method proposed by the TCG and is based on a digital signature made with the AIK on some data TPM-related: the content of the PCRs. The operation performed is called *TPM2_ Quote*. Along with the digital signature, a certificate validating the AIK used to sign the response is requested, in order to be sure that the AIK is owned by a specific TPM: the EK certificate is also sent in the TPM2_Quote.

In the remote attestation process it is possible to identify several roles, but the most important two are the Node, also called the Attester, and the Verifier. The Verifier is the one that wants to attest the trusted status of a node of the network, to do so it sends a challenge query, usually with a nonce in it for the freshness of the response, that is the *TPM2_ Quote*. The Node answers to this request by reporting its boot state, its configuration, its status and everything it's linked to its own identity.

The concept of identity is really important and needs to be better explained: the identities of a platform and of its TPM are a key point in the retorting phase of the attestation, since the Verifier needs to be sure that the response to the quote is coming from the targeted machine. Keys and certificates are the solution to this. As it was already pointed out, the TPM contains several key hierarchies and the one involved in the RA process is the *Endorsement Hierarchy* (EH). As each of the hierarchies, also the Endorsement one has a seed, a large random number which cannot be changed by the user, that is never exposed outside the secure boundary and from which the primary key, that is called Endorsement Key(EK), is derived. Primary keys like EK are generated using a FIPS-approved key derivation function (KDF), which hashes the primary seed together with a key template. The template for key generation is divided into two parts: the first part contains a description of the type of key that is needed (symmetric or asymmetric, for signatures or for encryption), the second part is used to introduce some entropy into the KDF. Usually the latter is set to all zeros as the TCG specifies, since we use TPM's source of entropy.

Starting from the version 2.0 of the TPM, the process of the generation of the EK is repeatable, this means that severals EKs can be created, by using the *TPM2_ Createek* command, but if the same seed and tampleate are used, the same key will be generated: the seed is the real cryptographic root. The seed is created by the manufacturer that can also create the EK pair: the private part of this pair never leaves the TPM while the public part is put into an Endorsement Certificate, which can be created with the command *TPM2_ActivateCredential*. This certificate attest the trustworthiness of that EK public part and also associate that key to an authentic TPM manufactured by the vendor. Different types of primary keys can be created (signing, storage, encrypting) using different algorithms (RSA, ECC, SHA-1, SHA-256) and they don't need to be stored in the NV-Storage of the TPM since the procedure of derivation is repeatable and the same seed with the same template results always in the same Key-pair.

So the EK and its certificate are extremely important since they are unique to each TPM constituting an important asset for privacy: for all these reasons they are used in a limited amount of cases and are used just as encryption keys, not signing keys. To obviate this, an Attestation Identity Key (AIK) is created starting from the EK and is used as signing keys: a user can decide to protect its privacy, avoiding traceability, by creating a different AIK for each one of the applications he is talking to. Then for each of these keys a certificate is created, attesting that the key is owned by a TPM that is authentic and manufactured by a certain vendor.

3.2.1 Activation of credential

The process of creating a key chain when the primary key is just used as an encryption key is not straight forward, but follows a process that is called *Activation of credentials* which also includes a Privacy Certification Authority(CA) to certify the attributes included in key's certificate: a third party is needed because it should be impossible to find any link between the several keys created by a TPM and the TPM itself. This operation guarantees that the attestation key belongs to a TPM with a certified primary key. So the primary key, that is the EK, is only an encryption key, not a signing one but signing keys are required so secondary keys are created (like AIK).

The CA creates certificate for those secondary keys that are then encrypted with the primary key public part and decrypted by the TPM with the corresponding private part, the steps are the following [10]:

CA SIDE

1. The TPM sends to the CA the certificate of the primary key(EK), that is the one inserted in the TPM by its manufacturer, and the features the credential (AIK) the TPM is requesting should have;

2. The CA verifies that the certificate is valid, usually checking directly with the manufacturer whether he has ever produced a TPM associated to that certificate;

3. The CA generates both a credential for the Key(AIK) and issues a certificate with the attributes specified in the first message. A secret, usually a symmetric key, to protect the credential is created too;

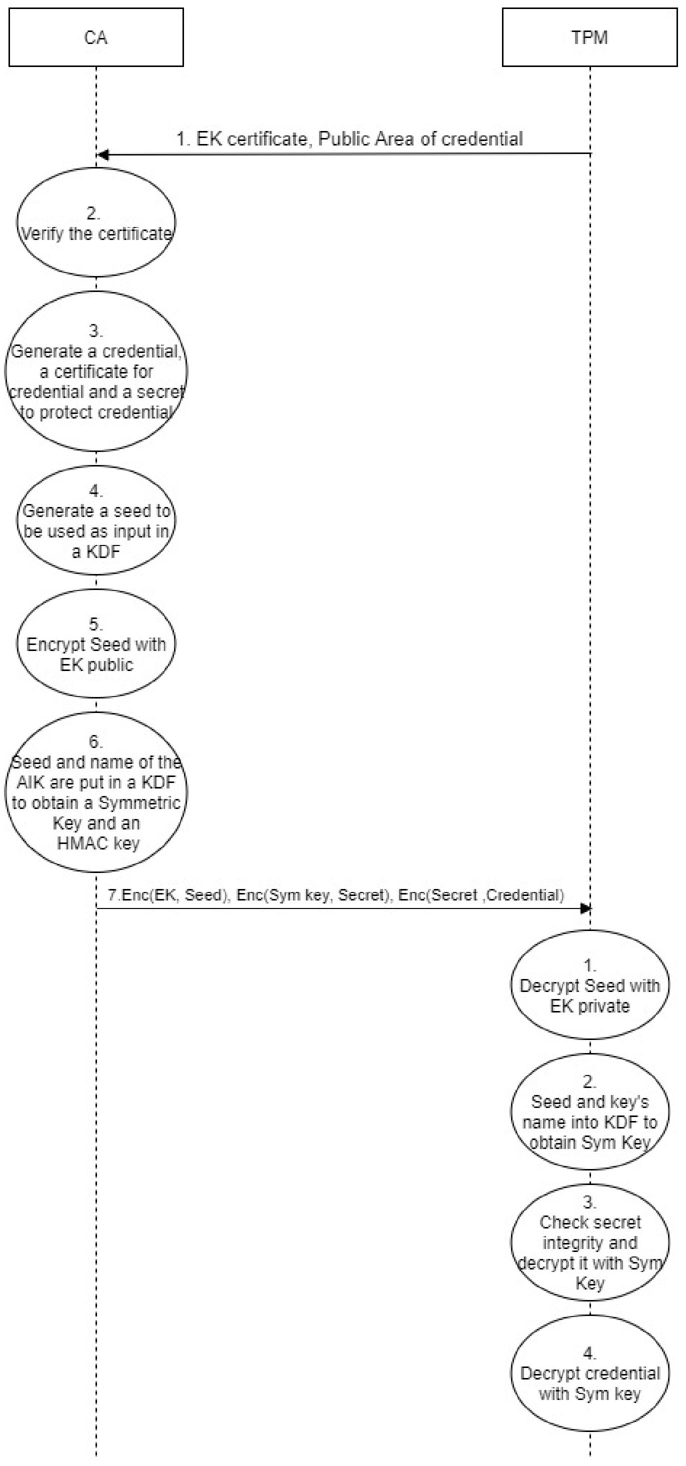

CA
TPM
1. EK certificate, Public Area of credential
2. Verify the certificate
3. Generate a credential, a certificate for credential and a secret to protect credential
4. Generate a seed to be used as input in a KDF
5. Encrypt Seed with EK public
6. Seed and name of the AIK are put in a KDF to obtain a Symmetric Key and an HMAC key
7.Enc(EK, Seed), Enc(Sym key, Secret), Enc(Secret ,Credential)
1. Decrypt Seed with EK private
2. Seed and key's name into KDF to obtain Sym Key
3. Check secret integrity and decrypt it with Sym Key
4. Decrypt credential with Sym key

Figure 3.4. Description of the activation of credentials protocol

4. The CA generates a seed as input of a KDF, the seed it's just a random number if the key is an RSA key, or something more complicated for an ECC key.

5. The seed is encrypted with the public part of the primary key(EK)

6. The Seed, together with the name of the key, is used in a TCG-compliant

KDF to generate a symmetric key and an HMAC key. The former is used to encrypt the secret, the latter conveys integrity.

7. The secret is encrypted with the Symmetric key just obtained and an HMAC is evaluated on it with the other key. The CA sends the encrypted seed, the encrypted secret with its integrity value and the credential(AIK) encrypted with the Secret.

TPM SIDE

1. The seed is decrypted using the private part of the Primary key(EK)
2. The seed and the key's name(AIK) are the input of the same TCG-compliant KDF function to obtain a symmetric encryption key and HMAC key. If the seed and the name are correct, the keys obtained are the same that were obtained by the CA and the TPM obtains the symmetrc key too.
3. The two key are used both to check the integrity of the secret through the HMAC and to decrypt it. So, if the integrity check is passed, the TPM obtains the secret.
4. The credential is decrypted with the Secret and it is activated.

So the result of this process is to obtain encrypted-user-data(seed) and a secret-encryption-key(secret) wrapped together with the integrity value and we also have the guarantee that the credentialed-TPM-object(AIK) is loaded on the TPM along with the public-key-object(EK), everything starting from the certificate of the EK and a set of desired features for the AIK.

3.2.2 Attestation process

Once the credential has been activated, before the attestation process can start, a few steps need to be taken. First of all the verifier needs to store TPM's public part of the Endorsement Key and of Attestation Key together with their certificates in its local database, where we can also find some policies including the expected initial state of the Attester(i.e. the boot_aggregate). All these measurements are stored in the local database and are used as reference in the process of attestation to determine the good state of the system (these are the Golden Values inside the whitelist). In order to be properly attested, a system needs to implement the so called *measured boot* that, if enabled, makes the system measure every software image during the whole boot process: in this way at boot time every software is measured and this value is stored both in a PCR, through the extension operation, and in an event log file where we can find one entry for each software measured. This event log is the file that was specified in section 3.1, that is /sys/kernel/security/ima/ascii_runtime_measurements.

After all these requirements are set up, the process can begin and this usually happens with the Verifier starting it: it could be done after a particular event happens or after a fixed interval or time in order to periodically check the state of the remote machine. The model we are going to study in the following is an example of how the process of remote attestation can be implemented, although this is not the only possible one, it is useful to identify some fundamental steps that are strictly needed.

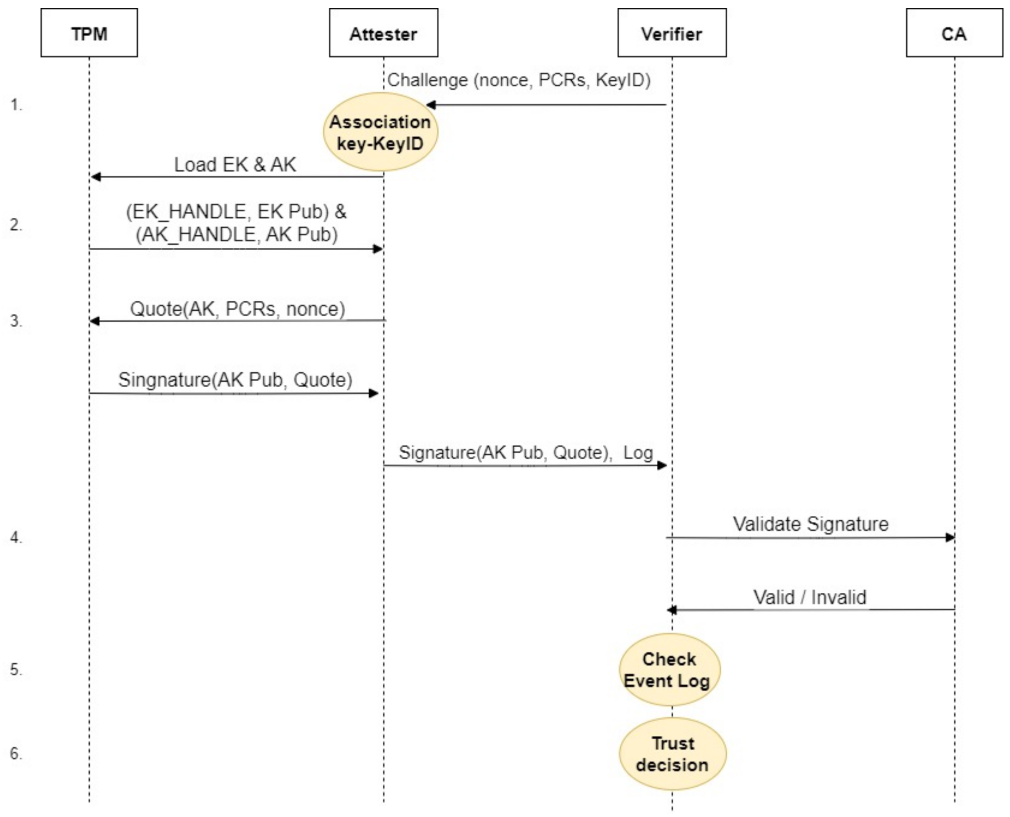


Figure 3.5. Description of the Remote attestation Protocol

Once the key provisioning process is over, the Verifier(V) asks the Attester(A) for its EK and AK certificates that it can verify against CA, to which the Attester is enrolled, so it knows its primary key (EK) certificate and can therefore verify the signing key (AIK): now the process can start [17]

1. V creates a challenge to be sent to A containing the list of PCRs it needs, a nonce for the freshness of A's response and the keyID, that is an ID associated to an EK specifing which is the TPM V wants to talk to;

2. A receives the challenge, recovers the corresponding EK and AIK and loads them if they weren't already in memory.

3. A runs the TPM2_Quote command asking the TPM to read the content of the PCRs asked by V and to sign it with the AIK, including the received nonce. What is actually signed by A with the AIK is a structure called TPMS_ATTEST, which contains some info about the key and the attestation process;

4. V receives the signature performed over the TPMS_ATTEST structure and verify it with the public part of the Attestation key, asking to the CA;
5. V validate the quote itself: starting from the information included into the TPMS_ATTEST structure, V reads the content of the log regarding the PCR asked and compare them with the Golden values stored in the local database. This comparison can be done value by value, analysing all the measurements done on each piece of software (os, bootloader of application) or can be done directly on the digest contained in the PCRs
6. V takes a decision whether the remote platform can be considered trusted or not.

Chapter 4

The ROOT Project

With the advent of 5G technology, accurate and secure time synchronization solutions are becoming increasingly important in the telecommunication field: although the requirements for synchronization have not become more stringent, the role of time synchronization has become much more critical [18]. Solutions that aim to ensure good results in time distribution networks are being studied in the **ROOT** research project (Rolling Out OSNMA for the Secure Synchronisation of Telecom Networks). One of the main topics on which ROOT's work is based on, is the evaluation of risks in time distribution networks [1], enumerating the cyberattacks that can be run against them and evaluating the related impact they would have on the entire architecture.

4.1 GNSS-based time synchronization solution

One of the emerging solutions for time synchronization, is the use of *satellite-derived timing information* into the process of provisioning an absolute time information in **Time-Sensitive Networks** (TSNs). The technologies on which this solution is based on are **GNSS** (Global Navigation Satellite System) services and transport domain networks: the use of GNSS receivers, terrestrial caesium clocks and a dedicated transport protocols, can satisfy the requirements on time synchronization accuracy and precision, which should be in the order of *sub-nanoseconds*. Among the GNSS services you can find *Galileo*, *GPS* and *BDS* that are offered by several space agencies. Inside these distribution of time networks, synchronization is obtained through a system of **Primary Reference Time Clocks** (ePRTCs), terrestrial accurate clocks based on the GNSS technology that are able to carry the time information across the network using a dedicated protocol, usually the **Precision Time Protocol** (PTP) or the **Network Time Protocol** (NTP). The former is thought for smaller network since it can satisfy more stringent requirements, being able to obtain an accuracy of microseconds and even nanoseconds [9]; the latter is designed for large and dynamic wide area networks obtaining an accuracy of some milliseconds [8]. The PTP protocol can distinguish among several types of clock: the source of the time information are the ePRTC clocks, which are the source of time of the **Grandmaster Clock** (GM) that gives the time reference for the whole network thanks to a system of ordinary clocks called *Slaves* coordinated by the GM [19].

4.1.1 The GNSS-based solution architecture

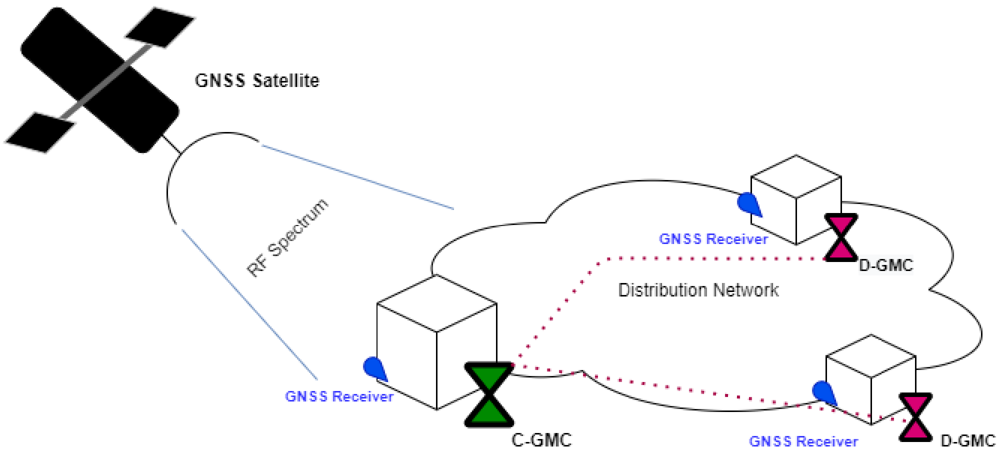


Figure 4.1. High-level architecture of a time distribution network based on GNSS systems, Centralized-Grandmaster clocks and Distributed Grandmaster Clocks

The goal of this solution is to obtain a network-wide time synchronization accuracy of *nanoseconds*: the reference time information is the one generated by a GNSS satellite which is then transmitted to GNSS receivers situated on Earth directly connected to the edge node of the distribution network. The time information is transmitted from the satellite to the receiver by means of *Radio-Frequency signals* (RF): that is why this area between the 2 devices is called *GNSS RF Spectrum*. The edge node, that is directly connected to the GNSS receiver, is also linked to a really accurate terrestrial clock made of iridium or caesium called *Caesium Atomic Clock* (Cs AC): these two different sources are combined together by the Centralized Grandmaster Clock (C-GMC) that generates an accurate time reference.

The C-GMC is the access point to the time distribution network since it supports specific transport protocol, such as PTP, NTP or WB-PTP, used to obtain synchronization between the GMCs and all Slave Clocks. In order to provide more robustness to the distribution network, in case the C-GMC either fails or is unreachable, a backup system is available in the network: multiple reference clocks called *Distributed Grandmaster Clock* (D-GMC) [20] are available. This devices, as well as the C-GMC, combine two different sources of time information to obtain a time reference: the first one is the information coming from the GNSS receiver, while the second one is from a device called **Ove-Controlled Cristal Oscillator** (OCXO) or **Rubidium clock** (RC), that are cheaper than the caesium or iridium ones.

In order to have an higher level of robustness, typically a network includes two C-GMC (a backup clock is available), as well as two level of D-GMCs, located in different physical spot but with a similar configuration. These devices are intrinsically hierarchical and this is due to the fact that they are employed in a layered

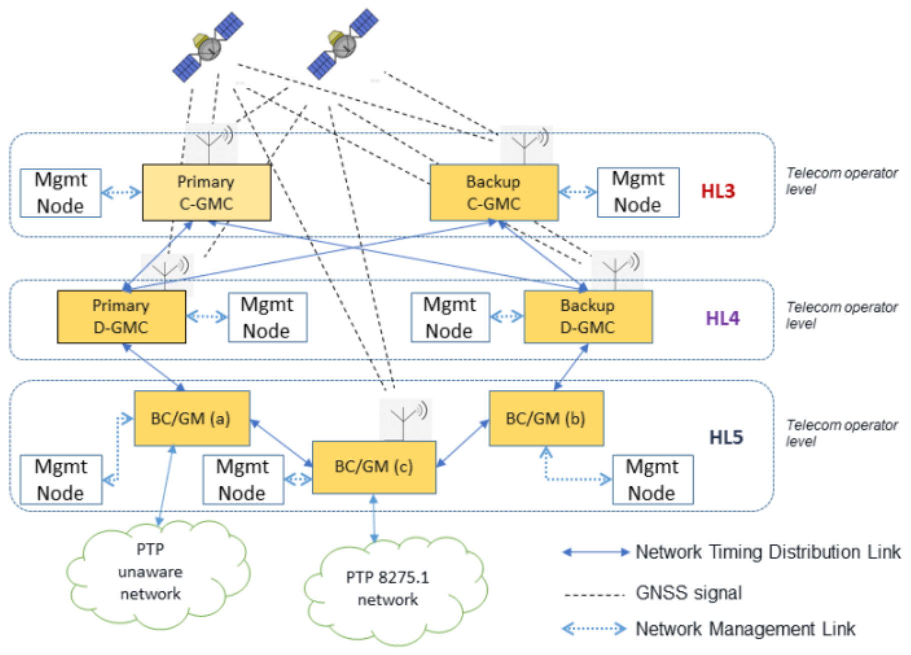


Figure 4.2. Hierarchical level organization of a time distribution network based on GNSS services (source [22])

way in telecommunication networks . An example of this architecture is the one that organizes the C-GMCs and D-GMC in three different operational levels(4.2) :

- *HL3*: the layer that is ideally the nearest to the GNSS-satellite. This is the regional level, the first to receive the time information. It contains the primary C-GMC and the secondary one in case of failure of the latter, but they are in different physical locations.
- *HL4*: this is the the layer in the middle, also reached by the signal of the GNSS-satellites. This is the metro aggregation level and contains the second level clocks, the D-GMCs; even in this case we have a backup clock to increment the resilience and robustness of the network. Both of the D-GMCs have a direct channel of communication (usually an optical fiber link) with both of the C-GMC.
- *HL5*: this is the last hierarchical level, the most distributed one and also the one directly connected to the PTP-unaware network and to mobile stations PTP-aware. The distribution devices employed in this layer are simpler than the ones of the layers above, they are called BC/GM and, as well as Grandmaster clocks, they contain a GNSS-Receiver, the difference is that there is no caesium or rubidium clock to support the receiver. This devices are directly linked among them and at least two of them are connected to the 2 D-GMCs of the layer above.

Each one of the time distribution device of every layer is directly connected to a *management node* (Mgmt Node), in order to be configured and do some troubleshooting in case of a failure or an attack occurs.

Probably the most suitable protocol for accurate time distribution in network including clocks organized in a hierarchical way like the one just described, is the PTP (IEEE-1588) and in particular its extension WB-PTP (White Rabbit - PTP [21]): thanks to this enhancement it is possible to obtain a precision in the synchronization in the order of nanoseconds or less.

4.2 Attacks against GNSS-based architectures

Studying the architecture of the GNSS-Based distribution network, seems pretty clear that the time information, that way in which it is distributed through the architecture and the synchronization among all the components are all key elements in 5G telecommunication networks. Thus they represent an important asset that is increasingly becoming target of cyberattacks. Distribution of a wrong time or position information, delays in the receipt of time information, time desynchronisation in the network and Denial of Service (DoS) are all possible result of a poor security management.

Analysing the network we can identify two main critical areas for security, which are the *GNSS RF Spectrum* and the *Time Distribution Network*: the former is the area that goes from the GNSS-Satellite to all the GNSS-Receivers while the latter is the set of Clocks, GNSS-Receivers and protocols used for the distribution of time. They are two very different areas as are the threats to them, but despite this they represent two possible weak points of distribution networks and thus they need an appropriate study and protection against attackers.

4.2.1 Classification of attacks against GNSS RF Spectrum

The GNSS RF Spectrum is the region that goes from the GNSS-Receiver that generates the time information and sends it using radio frequency signals to the GNSS-Receivers placed on the node that could be a GMC of a BC/GM. Obviously, for the node receiving the time information, on which the synchronisation of the entire network is based, it is very important that the signal can be considered trusted and that the sender is indeed the satellite and not any attacker: integrity and authentication are key point in this context.

It is possible to identify three main classes of attacks that can be run against the communication link between the GNSS-Satellite and the GNSS-Receiver [22]:

- *Meaconing*: this class of attacks consists in the retransmission of an old packet sent by the satellite to the node. Essentially the attacker stores the authentic packet and sends it to the receiver with a variable delay, in this way an old information is transmitted to the node and this undermine the overall Quality of Service.
- *Jamming*: in this case the attacker emits interference signals in order to disrupt the functionalities of the receiver. By sending RF signal, the attacker wants to block the reception of GNSS signal at receiver side.

- *Spoofing*: the goal of the attacker is to produce false information at the receiver by sending forged GNSS-signal with wrong time/position data [23].

Several studies have been conducted on this type of attacks and different solutions have been found: two are the security aspect to be taken into account, authentication and integrity. One of the most important solutions found is the one brought by the European Galileo Program which added authentication to the GNSS-signals. The solution is called **Open Service Navigation Message Authentication** (OSNMA) and, along with authentication, it also conveys integrity: its duty is to ensure that the signals received by a node of the distribution network was actually produced by a Galileo satellite and that it was not modified while in transit. By using this type of system, spoofing-like attacks can be easily detected.

4.2.2 Classification of attacks against the Distribution Network

The class of attacks that are presented in this section are the attacks that try to exploits the vulnerabilities of the set of nodes, GNSS-Receivers, Clocks and protocols that compose the time distribution network and aim to undermine the availability, integrity and accuracy on the exchanged time information. There are two major types of attack that can affect this portion of the architecture and they are:

- Attacks against the underlying protocol (usually PTP or its enhanced version WR-PTP) of for the distribution of the time information. Among them we find *replay or delay attacks, DoS, MITM*;

- Attacks against specific devices like the Distributed or Centralized GMCs, trying to disrupt them of gain access into the network both using hardware and software technique.

Attacks against the PTP protocol

The Precision Time Protocol, as well as its enhanced version the White Rabbit Precision Time protocol, are two of the most widely used protocols for time distribution. Since they are so much used, several ways to attack them have been found [24]:

- *Denial of Service (DoS)* can be carried out in several ways and at various network layers;

- *Man-In-The-Middle (MITM)* that includes delaying attacks, reply attacks, packet content manipulation, packet removal and also masquerade attacks : the aim is to ensure that the node receives incorrect or delayed information.

- *A new class of threats* has been found by a group of researcher: these threats consist of two variants of the DoS spamming attacks trying to temporarily steer of permanently skew the clock or worse to conduct a clock takeover attack, both of master and slave clocks [25].

It is worth pointing out that some mitigation techniques and protection solutions have been studied and proposed: first of all a new standard (**IEEE-1588-2019**), which contains a possible solution based on a multipronged approach, has been proposed. Another noteworthy solution to the PTP security problems is a modified version of the protocol based on the key management system of the NTP and on an identity-based authentication system [25].

Attacks against the software

Protecting GNSS-satellite signals in the RF Spectrum and during the exchange of information through PTP-like protocols in time distribution networks is necessary but it is not enough: another vulnerable point in the architecture of time distribution are the nodes themselves. *C-GMC, D-GMC and BC/GM*, that represent all the nodes of the network, are still vulnerable to potential attacks against the software running within them; this type of attacks must also be taken into account in order to achieve total network protection. These devices could be attacked and compromised by generating and exchanging incorrect time information, leading to poor synchronisation of the network nodes. The vulnerabilities that can affect this kind of devices are multiple and can be exploited in several ways:

- *Gain physical access* to the node installing on it some malicious code using an external storage or modifying some configuration options;
- *Exploitation of vulnerabilities* (like buffer overflows or other type of unchecked conditions) of the Operating System or other software installed on the node;
- *Installation of malicious code* inside the targeted node, also using other type of virus like worms or rootkits.
- Using *side channels attacks, software backdoor, memory scraping or software tampering* attacks to modify the configuration of the software running on the platform or the software itself.

According to the list just drawn up, both hardware and software security are really important to keep a node safe and secure. Hardware side, the protection is firstly physical, in fact the device could be physically damaged, all the sensors could be tampered and the platform could be subject to reverse engineering and be cloned. Software side, the attacks are multiple and aim to obtain a misbehaviour of the programs running on the node or to extract sensitive information. The most effective countermeasure to these kind of software threats are Trusted Computing and Software Attestation solutions, able to detect changes in the software configuration and code. This kind of techniques will be the main topic of next chapters.

In the figure 4.3 a possible Software stack running on the GMCs is provided [20]: of course this configuration is only a general description of the stack, which can vary greatly, and just wants to be a simplified example to let the reader understand which are the possible risks and the corresponding threats of a PTP node.

Of course a **GNSS-Receiver** is included in this stack configuration in order to receive the signals from different types of satellites (GPS, Galileo, BeiDou, and/or GLONASS) to estimate *Position, Velocity and Time* (PVT) information. The receiver talks directly to a Linux kernel support that can receive and process two different types of output:

- *NMEA 0183*: a textual interface providing the PVT data coded according to the National Marine Electronics Association (NMEA) 0183 standard;
- *1PPS*: an high precision analog signal with leading pulse edge synchronous with the beginning of each second of the time scale

These two inputs are processed by the **kernel** which make them visible to two different devices: a serial port, */dev/serial0*, addresses the NMEA input while the Linux Pulse Per Second Application Programming Interface (PPSAPI) is the one dealing with the 1PPS, and the device is */dev/pps0*. The data collected by these two devices are then passed to the **gspd** service daemon, that can parse GNSS data and then sends them to **ntpd**, the daemon implementing the NTP version 4. The communication between the 2 daemons takes place through two **shared memory** segments. The ntpd daemon is then able to correct and set the internal clock of the node, assuring the needed synchronization.

The gspd daemon and the shared memory segments could also be absent: in this case the device receiving the NMEA and 1PPS outputs are slightly modified and the data could be directly passed to the ntpd daemon, through two dedicated drivers.

After the internal clock has been properly synchronized, the ptpd (Precision Time Protocol Daemon) is adopted to distribute the time over the time distribution network.

This description of the SW, although general and not very detailed, shows how important it is to protect the nodes in the network, which are responsible for the information exchanged within it and therefore the synchronisation achieved there.

Other types of attack

As it was specified in the design of the time distribution network, each node is linked, physically or remotely, to a management device: attacks targeting the underlying network protocols used for the management of the time distribution devices are also possible. Among these protocols we can find the ones for the remote access like the *Secure Shell* (SSH) or *Transport Layer Security* (TLS), the ones for authentication like RADIUS, the *Simple Network Management Protocol* (SNMPv3) and classical network protocols like *TCP/UDP* and *ICMP*. All these protocols are usually prone to manipulation of security data configuration (like keys, certificates

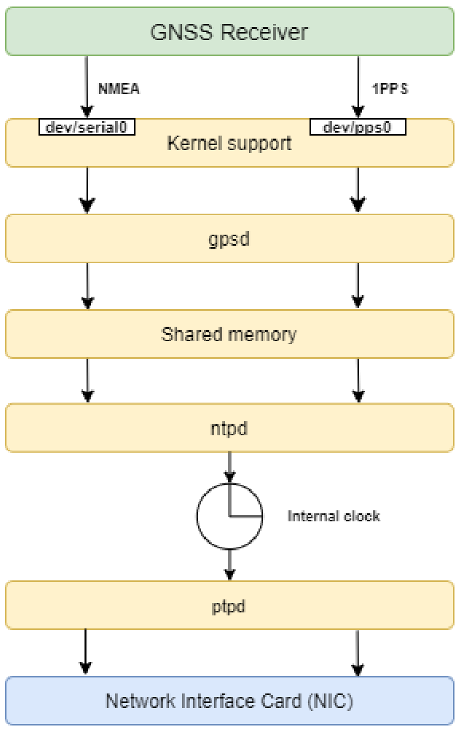


Figure 4.3. Possible organization of the Software Stack running on a GMC

or access rules) or of network configuration parameters, but also to misconfiguration. Typical attacks trying to exploit these kind of vulnerabilities are *DoS attacks, replay attacks and MITM*, aiming to give to the destination device wrong or delayed information, but also to obtain sensitive information exchanged between the GMC and the management node. Another class of attacks could be the network attacks, like ARP poisoning of spoofing attacks, the flooding attack or DNS attacks.

Nowadays several countermeasures exist against these well-known attacks, like the use of firewalls, IDS and/or IPS, packet filtering and the adoption of good network policies.

Chapter 5

The Keylime Framework

Developed by a security research group in MIT's "Lincoln Laboratory" and presented at the end of 2016 through the whitepaper *"Bootstrapping and Maintaining Trust in the Cloud"* [26], the Keylime Framework aims to provide high scalability to the remote boot attestation process, proposing runtime integrity measurement solutions. It is now a CNCF open source project which helps both users to continuously check remote nodes using a hardware-based cryptographic root of trust and developers by providing them a simpler way to manage the technology of the TPM 2.0.

5.1 Background

The context in which the idea of the Keylime framework was born is that of Cloud Computing and, more specifically, *Infrastructure as a Service* (IaaS): we are therefore talking about a network scenario where there are several nodes connected to each other, usually managed by a single tenant. So the final user is provisioned with resources, in the form of computational power, storage and/or connections, that he can use to run his own software. In this kind of networks, resources consist of *cloud nodes* that could be in the form of physical or virtual machines: this resources are given to the user who is able to deploy his software to these nodes and control them. The key point is that tenants have no control over the underlying infrastructure so they are not able to ensure, with their own implementation, that the platform given by the IaaS provider remains in a good and safe state during the computation. Nowadays, the IaaS cloud services available do not provide any effective method to check the integrity and the trustworthiness of nodes and of the environment in general. The policies currently in place in these infrastructures severely restrict tenants' ability to establish unique and unforgeable identities for individual nodes that could not be tied to hardware-based root of trust but only software-based solutions can be adopted. This would force tenants to send through the cloud provider's network unprotected information and sensitive data.

When TPM came out, it seemed to be a good solution to the problem of establishing a trusted hardware root of trust, conveying a unique identity to each machine by referring to objects like the Endorsement Key, but these hopes were soon dashed. The cloud environment we are now considering, the one of the IaaS, doesn't fit very well the main features and ideas of the TPM: first of all the hardware/physical nature of the chip is at odds with the trend in cloud environments towards virtualisation, TPM's standards and their implementation tend to be too complicated and, not least, its low performances, requiring more than 500 ms for a digital signature.

In order to deal with these issues, the developers of the framework have outlined some basic security features that a IaaS should have to be compliant with Keylime [26]:

- *Secure bootstrapping*: a tenant should be able to securely inject a root secret into all of his nodes and, starting from this one, derive more secrets;
- *System integrity monitoring*: a tenant should be able to monitor each of his node, being updated regarding integrity deviations of the underlying platform within a second;
- *Secure layering*: a tenant should obtain secure bootstrapping and system integrity monitoring also for virtualized nodes by leveraging a TPM in the provider's infrastructure;
- *Compatibility*: a tenant should be able to use hardware-rooted cryptographic keys in software to secure services that they already use (e.g. disk encryption);
- *Scalability*: it should be possible to meet the above requirements in an IaaS system with thousands of virtual resources.

In the development of Keylime, which can be considered as an end-to-end IaaS trusted cloud key management service [26], all these requirements were taken into account and were met by the researchers: in order to satisfy the requirement about the secure bootstrapping, they developed a new **bootstrap key derivation protocol** for installing root secret and injecting identities into nodes. They also implemented a way to have a **periodic remote attestation process** monitoring the trusted status of the cloud node, by linking the attestation to the identity of the remote platform, obtaining in this way the System Integrity Monitoring required. This two solutions are provided both for virtualized environments and bare-metal, which means that *virtualization support* is given, while the compatibility with the most common services in the IaaS field (e.g IPsec, Puppet, Vault,LUKS) is granted since they were integrated in the framework. One of the most appreciated features of Keylime is its proven **scalability** since it is able to manage thousands of virtual nodes at the same time and to process thousands of integrity report (IR) per second.

Keylime's main objective is to decouple the *identity bootstrapping* from the *management of identities*, and this goal is reached by putting itself between the software based cryptographic Services and the TMP itself, exposing a simple interface to developers (as shown in figure 5.1). In this way Trusted Computing and high-security services work together but independently: the former manages the bootstrapping of the keys and the latter manages the identities.

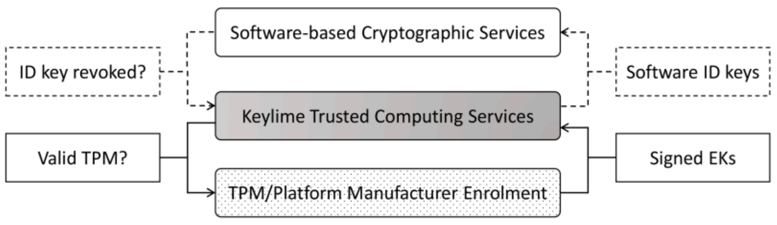

Software-based Cryptographic Services
ID key revoked?
Software ID keys
Keylime Trusted Computing Services
Valid TPM?
Signed EKs
TPM/Platform Manufacturer Enrolment

Figure 5.1. Keylime as a middleware between software-based Cryptographic Services and trusted hardware. Decoupling of identity bootstrapping from identity management

5.2 Design

In the development of the framework, researchers tried to address the limitations of the cloud environment: the goal was to combine trusted computing and IaaS in order to obtain a hardware root of trust to be used as starting point to build trust in the cloud infrastructure, considering both the provider's systems and tenant's elements.

During the analysis of risks and threats, some assumptions were made: the cloud provider is *semi-trusted*, which means that he is trustworthy but there could still be some malicious insiders in its organization and some part of his infrastructure could be under the control of the adversary. Specifically, the administrator has in place some control systems and some policies to mitigate an eventual attack, while the adversary is assumed to be able to monitor and modify portions of the network or of the storage. Adversary's goal is to obtain persistent access to a tenant system to steal, modify or delete tenant's data or services. To obtain one of these, the attacker should modify the code of the process running but such modifications should be detected by the runtime measurements. What the adversary is not supposed to be able to do is to physically tamper with any host's component or system.
What researchers also assumed, is that the administrator of the cloud infrastructure is not voluntarily deploying any malicious code in the infrastructure and also that the TPM is configured in the correct way creating valid credentials.

5.3 Keylime nodes' architecture

As already pointed out, Keylime offers virtualization support in order to be compliant both with physical node and virtual nodes; this work focuses on architectures containing physical nodes.

It is really important to highlight that one of the main goals of Keylime is to protect Tenant's sensitive data (e.g identity keys or TPM credentials) from the provider of the IaaS. This is the reason way in the first phase of the bootstrapping process a symmetric ephemeral key K_b is generated. This key is used to encrypt the data d that are passed to the node, using the AES-GCM algorithm($\mathrm{Enc}_{K_b}(d)$). In this way, when the node performs the registration to the framework, this K_b is associated to its UUID (and also to its IP address). This payload, containing sensitive data *d*, doesn't have a fixed structure but can contain several type of data: software identity and certificate for high level services, some scripts to be run when the payload is decrypted or when a revocation notification is received, a revocation certificate to verify the notification or other files related to the revocation service.

The framework's simplified architecture is presented in figure 5.2 and presents several components:

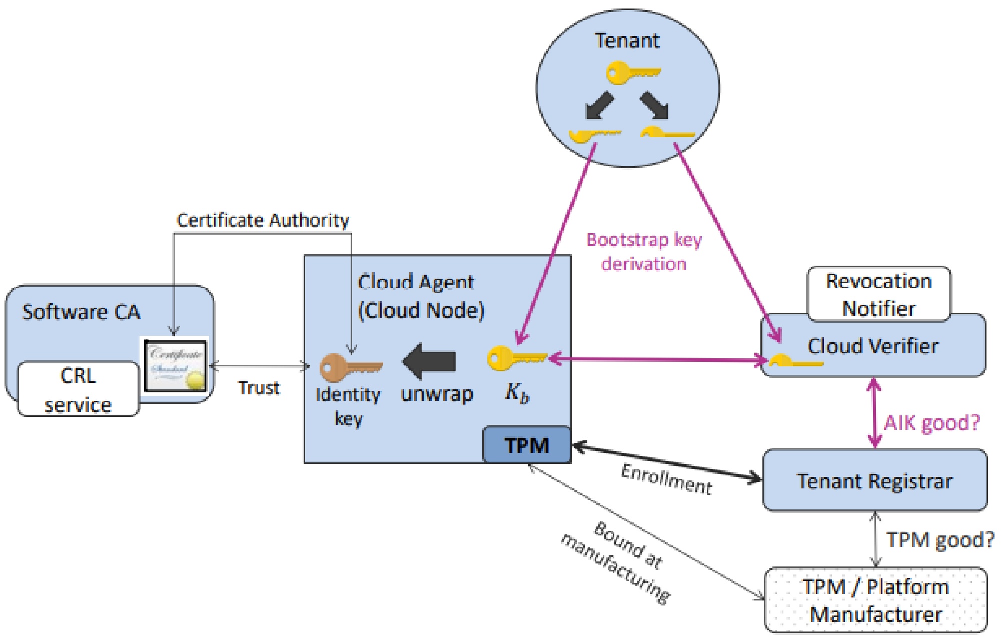


Figure 5.2. Keylime simplified structure (no virtualization support) (source [26])

- The **Registrar** is the component to which every node of the network has to register itself using its own *UUID*, an alphanumerical identifier. For each node registered, the Registrar stores three different pieces of information: its EK_{pub} keys, its AIK_{pub} keys and its EK certificate. In this way the Tenant has the possibility to associate an EK_{pub} and an AIK_{pub} to a node, which holds the corresponding EK_{priv} and AIK_{priv}. The verification of the EK certificate is not a Registrar's duty but a Tenant's one and it is performed during the *key derivation protocol*. So the Registrar is not storing any Tenant secret;

- The **Cloud Verifier (CV)** is the core element of the whole framework: once a node is registered to the Registrar, the Tenant can start monitoring it by asking to the CV to verify its integrity state. For this reason, it has a central role in the whole architecture since it is the element that actually checks the integrity state of the components of the infrastructure. It relies upon the Registrar for retrieving the AIK_{priv} that is necessary to validate the *TPM_quote* from the node;

- The **Cloud Agent** is the software component that runs on the remote node whose integrity status the Tenant is interested in;

- The **Tenant** is the customer which is using the IaaS: his goal is to verify the status of its nodes. He starts the framework and ask to the Verifier to start/stop monitoring a node. When the verifier is asked to verify the integrity state of a platform by the Tenant, an encrypted payload d is also sent to it, containing basic information to perform the integrity check on the node;
- The **Software CA** is responsible for combining trust and TPM-based integrity measurements together with the higher-level security services (like IPsec), in this way it is not necessary to make services trusted computing-aware;
- The **Revocation Service** allows the system to react when the trust state of a node is not verified. When the Verifier finds an anomaly in the IR of a node, a notification is sent to the CA and to all the other nodes interested in the compromised service. Upon receiving this notification, the CA updates its own CRL by revoking the certificate related to the identity key of the untrusted node, then publishes the updated CRL. After consulting the CRL, each node can react to the notification in a different way, by executing a script the Tenant wrote for it.

Just by looking at all the components of the framework, it is possible to understand that they are all linked together and that one cannot work properly without the others. Anyway, it is possible to identify 4 operational phases in which the process of bootstrapping and maintaining the framework is divided, these phases are:

1. *Node Registration Protocol*;
2. *Three Party Bootstrap Key derivation Protocol*;
3. *Continuous Remote Attestation*;
4. *Revocation Protocol.*

The Node Registration Protocol

The first step in the bootstrapping process of Keylime is the creation of the Registrar, that is trusted by the Tenant after a verification of its identity. Then, when a new node has to be initialized and added to the framework, its AIK needs to be validated: to do so, the so called *Node Registration Protocol* has to be performed (Figure 5.3).

The protocol is compliant with the existing TCG standard for creation and validation of AIK keys and begins with the new node contacting the Registrar to register itself to the framework with its TPM's credentials. The sequence of messages is the following:

1. The Node N sends to the Registrar R its identity (UUID) and its TPM credentials, which are the public part of its AIK (AIK_{pub}) and the public part of its EK (EK_{pub}).

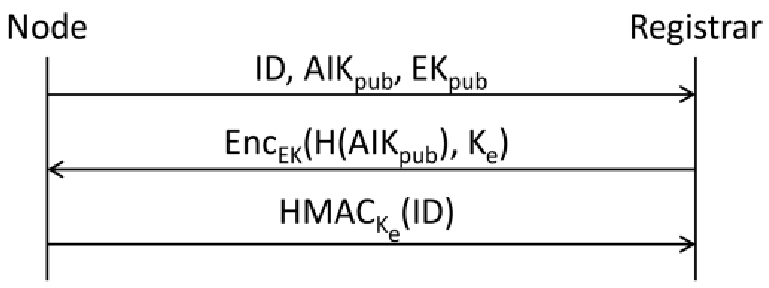


Figure 5.3. The 3 fundamental steps of the Node Registration Protocol (Source [26])

2. R receives $\mathrm{EK_{pub}}$ and checks its validity by contacting the manufacturer of the TPM, then, if it is valid, R creates an ephemeral symmetric key $\mathrm{K_e}$. This key is encrypted, along with the hash of $\mathrm{AIK_{pub}}$, using the $\mathrm{EK_{pub}}$ key. Doing so, R is creating a challenge in order to see if N actually owns the private part of EK ($\mathrm{EK_{priv}}$). In fact N will be able to read the ephemeral key if and only if he has the corresponding $\mathrm{EK_{priv}}$.

3. N receives the challenge by R and runs the *TPM2_ActivateIdentity* command, passing the encrypted payload to it. In this way, if $\mathrm{EK_{priv}}$ is the right one, the payload is decrypted, the ephemeral key is obtained and N can send a proof of its identity to R by sending the HMAC of its UUID computed with $\mathrm{K_e}$. Upon receiving the response by N, R evaluate the HMAC of the UUID obtained at the first step and, if the result is the same as the message just received by N, the credential are verified and the new node is activated.

The Three Party Bootstrap Key derivation Protocol

As already pointed out, the Tenant has to exchange sensitive data with the node so, once the Node has been successfully registered to the Registrar, it is necessary for the Tenant to make a key agreement with it. The idea is to obtain a bootstrap symmetric key $\mathrm{K_b}$, that can be used to encrypt the traffic between them. This protocol is called *Three Party* because it involves three different components: the Tenant, the Cloud Node and the Cloud Verifier.
To start this process, the Tenant generates a new random symmetric key $\mathrm{K_b}$, that will be used to encrypt data using the AES-GCM algorithm and then will divide this key in 2 different parts: **U** and **V**. The V part is a secure randomly generated number of the same length as $\mathrm{K_b}$ that will be shared directly with the CV, to be further provided to the node upon verification of its integrity state. The U part is generated starting from V by evaluating $\mathrm{U} = \mathrm{K_b} \oplus \mathrm{V}$ and is then sent to the Node. This division operation is done since the Tenant doesn't want the provider to obtain the symmetric key.

The protocol could be divided in three different phases, each one involving the three parties in a different way as it is possible to see in figure 5.4.
Phase A sees the Tenant and the CV exchanging only one message: it starts with the Tenant connecting to the CV over a secure channel to inform it that a new Node is available. The Tenant sends to the CV some information about this new

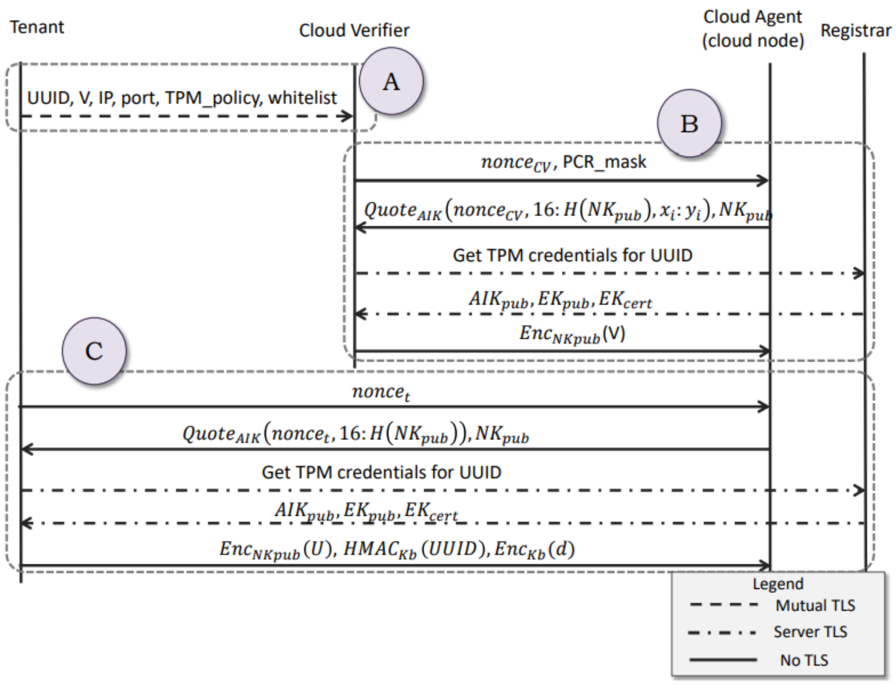

Tenant
Cloud Verifier
Cloud Agent (cloud node)
Registrar
A
UUID, V, IP, port, TPM_policy, whitelist
B
$nonce_{CV}$, PCR_mask
$Quote_{AIK}(nonce_{CV}, 16{:}H(NK_{pub}), x_i{:}y_i), NK_{pub}$
Get TPM credentials for UUID
$AIK_{pub}, EK_{pub}, EK_{cert}$
Enc_{NKpub}(V)
C
$nonce_t$
$Quote_{AIK}(nonce_t, 16{:}H(NK_{pub})), NK_{pub}$
Get TPM credentials for UUID
$AIK_{pub}, EK_{pub}, EK_{cert}$
$Enc_{NKpub}(U), HMAC_{Kb}(UUID), Enc_{Kb}(d)$
Legend
Mutual TLS
Server TLS
No TLS

Figure 5.4. Three Party Bootstrap Key Derivation Protocol in 3 phases: A, B and C (Source: [26])

Node: its *UUID*, the *V* part of K_b, the *IP address* and the *port* to which the Node is reachable, the *TPM policy* and the *whitelist*. These elements are really important for the attestation process since the TPM policy specify which are the PCRs to be read when a TPM Quote is sent and the values they should contain, the whitelist contains the trusted digest of the programs and files stored at the Node, whose value has to be validated against the IMA Measurement List sent by the Node with the quote operation.

Once phase A is completed, the two *phases B and C* start in parallel: in the former the attestation process is carried out by the Tenant and in the latter it is carried out by the CV. These phases are done in parallel so, at the end of them, the Node obtains U from the Tenant and V from the CV. The idea is always the same: the Tenant doesn't want the provider of the cloud service to be able to read this communication, so the Node, which doesn't have a certified software identity, generates an asymmetric key pair *NK* to encrypt sensitive information. The NK_{pub} is then sent to both the Tenant and CV to have an encrypted communication during the two phases. The authenticity of the public key is proven by extending NK_{pub} into the PCR 16, that is then further sent to both the entities in a Quote operation: this means that the ephemeral key is authenticated using TPM credentials.
Phase B starts with the CV sending to the Node a freshly generated *nonce* (n_{CV}) and a *PCR mask* containing the PCRs that the CV wants the Node to include in the next Quote operation. The Node replies with the requested Quote: it is evaluated over the PCRs requested by the CV and their values, the value of the PCR 16 to validate the ephemeral key and the nonce just received. Along with the Quote, evaluated using Node's AIK, the NK_{pub} key is sent. Upon receiving this message, the CV gets the TPM credentials of the Node, using a server authenticated TLS

with the Registrar: in this way it is able to decrypt the Quote and to verify the trusted state of the platform, by checking the nonce and all the values of the PCRs. By decrypting the Quote, the AIK is automatically validated by the CV and the identity of the node is verified. The last verification is made on the NK_{pub} : the hash of the received value is evaluated and it is compared with the content of the PCR 16. If every verification is successful, the last message of the phase is sent by the CV: the value V encrypted using NK_{pub} is received by the Node.

Phase C contains the same sequence of messages of phase B with some small differences: the Tenant doesn't sends any PCR mask in the first message since the Quote of the second message is evaluated only on the nonce and on PCR 16. This is due to the fact that the Tenant doesn't perform any kind of integrity check of the Node, its only goal is to verify the identity of the Node and to validate the freshly created NK_{pub}. In the AIK validation phase, the Tenant performs some verifications on the EK certificate: it has to be issued by a trusted TPM manufacturer whose certificate should be stored in a Tenant's local repository, the public key contained in the certificate should the same as the EK_{pub} received by the Registrar and the signature on the certificate has to be authentic and this is verified using the public key contained in the TPM manufacturer's certificate.
The last difference is in the final message: along with the encrypted value of U (of course this time is U and not V), the HMAC of the UUID, evaluated with K_b, is sent. The goal of the HMAC is just to let the node verify the correctness of K_b, that now the Node is able to compute by combining the just retrieved U and V. Once the HMAC of the UUID is checked and K_b validated, the Node use this key to decrypt $Enc_{Kb}(d)$, the encrypted payload received at the beginning of the bootstrapping phase. When the data *d* are retrieved, the Node has to delete K_b and V while stores U in the TPM NVRAM in order to be able to go through the bootstrapping process again if rebooted.

The Continuous Remote Attestation

At this point we have seen the first two phases of the bootstrapping and maintenance of trust process in a cloud environment with Keylime: so far we have seen how a node can register itself to the framework by using its TPM credentials and how the Tenant can verify that the Node is in a trusted status. The next phase is to periodically check that the Node is still in a trusted state by performing a *Continuous Remote Attestation* operation. The Tenant can specify an attestation interval in the configuration file and the Verifier will poll the cloud Node to monitor its integrity state every time this interval expires. This periodical verification of the state is done by checking the IMA Integrity Report that the Node sends, once time expires, to the Verifier within the Quote.
The attestation process is shown in figure 5.5: the CV periodically requests a Quote to the Node and performs a check on it so he can detect every integrity violation in the system.
The checks actually performed by the CV are:

- The Quote signature should be valid and the signature key should be the AIK_{priv} key provided by the Registrar;

.

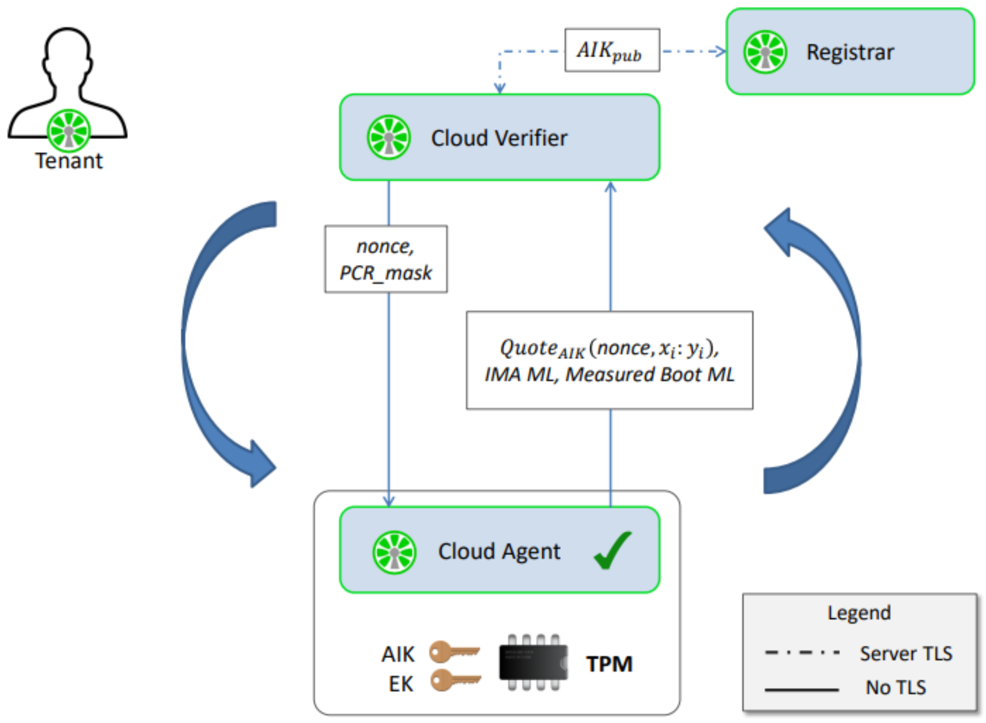


Figure 5.5. Periodical Remote Attestation Process performed by the CV

- The Quote should contain the PCRs specified in the TPM Policy and the unspecified PCR 16 (containing the digest of the NK_{pub});
- The IMA ML should be validated and should match the PCR 10;
- The processes and files contained in the IMA ML are the one specified in the whitelist and their digests match the one in the whitelist too.

It is worth highlighting that the ML is not explicitly signed, but the integrity of the ML is performed implicitly: the quote, which is signed and therefore its authenticity is guaranteed, contains the value of the PCR 10 and the ML should match this value. This mean that, when the Verifier receives the list with all the measurements, it performs the extension operation with each value of the list and at the end, verifies that the resulting value matches the value of the PCR 10 contained in the Quote. In this way the integrity of the ML is verified and its values can be checked against the whitelist.

Timing is a very relevant factor in this context: of course there is a lower bound for the time interval that can pass between an attestation and the other. This is give by the time needed to perform the attestation itself which is usually near to 1 second, since just the *TPM_quote* operation lasts about 500 ms. By default this interval is set to 2 seconds, but as already said, it can be set by modifying the configuration file: however it is a very important parameter since it also define the amount of time an attacker have to perform an attack without being detected.

The Revocation Protocol

During the Remote attestation process could happen that the integrity check reveal an integrity violation and the Node would not be considered trusted any more: the result is the CV triggering the *Revocation network*. This relies on the revocation notifier that is a server created by the CV at start up, implementing the publish/subscribe pattern. This is triggered by the CV sending a signed message, the *revocation event*, to this server that forward it to all its subscribers which could be: the CA that can react to this event by revoking the certificate of the identity key of the interested node, the cloud agents that can react by running their own revocation script, if any, or any other type of service interested in the trust state of the network.

5.4 Keylime and the ROOT project

It is important to remember that this work is to be intended in the wider context of the *ROOT project*, performing an analysis of risks and threats affecting time distribution networks. As already specified, the focus of this analysis is on the threats affecting only the nodes of the infrastructure, without considering all the other aspects, like the channels of communications. So the idea is to analyse the risks that the information takes the moment it has arrived at the node and not while it is in the network (as the network is something that can be trusted).
Referring back to the description of the ROOT architecture given above (in section 4.1.1), it is possible to understand that the distributed nature of this architecture makes it really similar to a cloud environment or to an IaaS, and the same similarity can be seen in the risks they both run. One of the possible risks that such an architecture may run is that of having one of its nodes compromised, be it a C-GMC or a D-GMC. This risk may result either in a simple modification of one of the node's daemons, for example the *ptpd*, or in an attacker taking over the node. It is then pretty straightforward that the just described *Keylime Framework*, being perfectly suited to cloud environments, may also be suitable for a hierarchical and distributed architecture such as that of ROOT.
Analysing Keylime's main features, it is possible to see how this framework seems a perfect solution for performing remote attestation also in the ROOT infrastructure [26]:

- Keylime offers a support for *System integrity monitoring*, performed through the continuous remote attestation process. This system could be used by ROOT's Tenant, along with the *revocation framework*, to be constantly updated about the integrity status of all the nodes of the infrastructure;
- Keylime offers a good *scalability*, being able to monitor thousands of resources at the same time. In this way, the characteristic large size of the ROOT network would not be a problem;
- Keylime results to be *compatible* with the major high-level security services, like IPsec, LUKS or Puppet, so it could be easily integrated in the whole system;

- Keylime performances, especially in terms of latency and quote per second (we are talking about ROOT that is a time distribution network), are pretty good. Taking into consideration physical nodes, Keylime was tested to be able to perform approximately 2500 quotes per seconds, having a bootstrap latency of about 750ms, deriving a key in less then 2 seconds and requiring as little as 110ms to respond to an integrity violation.

All these features makes us consider Keylime a perfect candidate to add bootstrapping and maintenance of hardware-rooted trust and continuous remote attestation in ROOT's systems. In the next chapters, the solutions adopted to install and use Keylime in the ROOT project will be exposed.

Chapter 6

Keylime Continuous Remote Attestation of a physical node with a physical TPM

The idea behind this thesis is to be able to run Keylime in the scenario defined by the **ROOT project**: an IaaS environment in which is it possible to perform the Continuous Remote Attestation process on both devices with a physical TPM and devices without a physical TPM install on them.
This chapter exposes the proposed solution for performing Remote Attestation in an architecture involving only platforms with a physical TPM installed on them. It will be described the physical prototype on which the installation of the *Keylime Agent* was made, along with all the commands and operations performed on it, the modifications made to the framework and the reasons why. The idea is to provide a sort of step-by-step guide and description of how to use Keylime to perform the system integrity monitoring: both the deployment of the *Keylime Agent* to the node to be attested and the installation and bootstrapping of the *Keylime Tenant*, *Keylime Registrar* and *Keylime Cloud Verifier* components will be described.

6.1 Prototype

In order to be close to the actual structure of the ROOT network, which involves very simple and low performance nodes, this work was in part carried out on a prototype that reflects these characteristics. The assumption that was made in this first phase of the experimentation is that all the physical nodes of ROOT's network have a *TPM* on board, so that there is no need to find alternative ways to use the framework, like virtualization or using a software TPM emulator. Obviously this is a very reductive assumption, limiting the number of possible cases very much, but it represents the first phase of experimentation.

The *Proof-Of-Concept Trusted Platform* represented in figure 6.1 is based on the *Raspberry Pi 4 single-board computer*, equipped with an *Infineon TPM v2 via an Iridium evaluation board* and preconfigured with the user space community TC

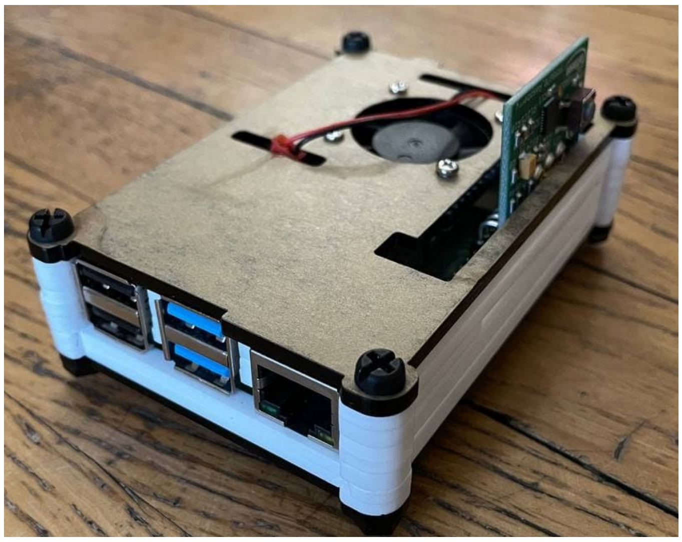


Figure 6.1. Photo of the physical Proof-Of-Concept Trusted Platform

software, IMA security subsystem enabled and a secondary partition encrypted with the key sealed to the *TPM's PCRs 0-9*. A complete list of prototype's characteristics is available in the Appendix A.
It is really important to point out that this kind of prototype should never be use in a production environment since it presents some serious criticality: the TPM chip could be easily unplugged, the electric signals could be easily monitored for attacks, but mainly it doesn't present any hardware *Root-of-Trust*, which means that the **CRTM** is missing. As reported in section 2.2.3, the CRTM is the first link of the chain of trust, an immutable set of lines of code, usually directly burned in the motherboard or written into the BIOS, which has the duty of checking that the right bootloader is read and run. Of course, the fact that the CRTM is missing makes the whole system *untrusted*, since we have no guarantee that the authentic bootloader was actually run. But it is important to underline that the untrusted state of the Proof-Of-Concept, due to the lack of a CRTM, doesn't invalidate in any way the operations done after the bootloader is run: this means that, assuming to have a properly working CRTM, the operations to be performed in order to be able to remotely attest a device would be the same.

6.1.1 Prototype bootstrap process with u-boot

As described earlier, the prototype doesn't contain any CRTM, either burned in the motherboard or written in the BIOS, that can ensure to the user that the right bootloader, in this case *u-boot*, was run. This is the reason why this prototype cannot be use in a production environment. In the following, it will be assumed that the right version of the u-boot bootloader is always launched.
The *u-boot* version (*2020.04*) installed on the board was patched in order to support some of the *TPM2_ Tools* commands, useful to start up the TPM, measure the boot components and extend the PCRs with these measures. This patched version

of *u-boot* is able to measure boot files, some configuration files and extend, following some directives, the specified PCRs. Which PCR each file's measure should extend is not defined by *u-boot*, it depends on the personal implementation and it is described in the Table 6.1.

PCR	**Measured Files**
0	u-boot.bin (u-boot image)
1	boot.src (boot script file, if any)
2	[N/A]
3	[N/A]
4	zImage/uImage (Linux kernel image)
5	platform configuration file (if any, e.g. config.txt)
6	additional parameters for Linux kernel command line (if any, e.g cmdline.txt)
7	[N/A]
8	[N/A]
9	[N/A]

Table 6.1. List of PCRs and related measurements by which they are extended

The last operation performed by the u-boot bootloader is launching the kernel but, as it possible to see in Table 6.1, PCR 4 has to be extended with the measurement of the kernel image: this is done by the bootloader just before lunching it. Once the kernel is run, the **TSS** starts interacting with the TPM and the IMA kernel module begins making its measurements, appending them to the *IMA Measurement Log*(ML) and extending the PCR 10. The whole process, from the moment in which the system is switched on to the moment in which the kernel starts, is called **secure boot**. Assuming to have a CRTM, if the system performs correctly all the steps just described, the *secure boot* can ensure that:

- The right bootloader was executed;
- The kernel just launched was measured;
- The measurement of the kernel was used to extend PCR 4.

What we cannot be sure of, is the fact that the right version of the kernel was run: the operation that checks the integrity of the kernel is called **authenticated boot** and leverages the measurement made on the kernel (that extended PCR 4). There are several method that can be adopted to implement the *authenticated boot*, probably the three most common are:

1. The first and most complicated solution is to encrypt the whole disk, including the partition containing the kernel, and decrypt it if and only if the values

of the PCRs 0-9 are the expected ones: a sealing operation is therefore performed. This means that the key used to encrypt the disk, which is stored in the TPM, is sealed with the first ten PCRs and that it can be unsealed only if, when the kernel is about to be run, the PCRs 0-9 contain the expected values (which are the same values they had at the moment of the sealing). The problem with this solution, which would provide the higher level of security, is in the version of u-boot: the commands related to the operations of *sealing* and *unsealing* must be authenticated but the few commands that is possible to add to u-boot are not authenticated. So the solution would be to patch the commands contained in the TPM2 Tools, implementing authentication for these commands: of course this would be heavy and time consuming operation.
This problem does not arise if the commands are run after the kernel started: in this case the *TSS* is started too and the commands we need to perform the sealing and unsealing phases are authenticated by it.

2. The second solution tries to overcome the problems found in the first one: if it is too hard to use authenticated commands before the kernel starts, let use them after the kernel is run, when TSS is available and authenticated commands too. Of course in this case it is not possible to encrypt the whole disk since, in order to run the kernel, the portion of memory containing its code has to be accessible. So the idea is to encrypt only a **secondary partition** that the kernel tries to mount as soon as it can, by default. In this way, if the kernel is not able to mount the partition because it was not able to unseal the key and therefore to decrypted the second disk, the operative system will send the system in *Emergency mode*, making the platform unusable. In order to implement this solution, the encryption key is sealed with the PCRs 0-9. When the kernel tries to mount the partition it also tries to decrypt it: to do so the decryption key has to be unsealed, but this operation could be done if and only if the PCRs from 0 to 9 contain the expected values. Of course this solution has a lower security level than the first one, because the kernel is not protected by the sealing and could be modified avoiding the mounting of the secondary partition, but it is way easier to be implemented and it is compliant with the requirements of the project.

3. The third solution is the easiest one but also the one with the lower security level. If our intent is to perform the remote attestation process, sooner or later the **IMA ML** will be sent to the attester that will check its integrity (this will be done with the Quote operation). The integrity of the IMA ML is checked by comparing every digest it contains with some *golden values* contained in Verifier's local storage, then the verifier checks that the hash of the whole log is the one expected. The first entry of the IMA ML is the *boot_aggregate*: when IMA starts, the first operation performed is to compute this *boot_aggregate*. This value is computed by hashing the PCRs 0-7, then this value extends the PCR 10. This means that, when the attestation process is performed, the values of the first eight PCRs are checked when the boot_aggregate is compared with the corresponding golden value: if they aren't equal, the attestation fails and this means that something in the boot

process was modified. So by performing the first remote attestation, checking the validity of the quote, we are indirectly performing the last step of the *authenticated boot*. This is the weakest solution since the boot process is not verified until the first attestation is performed: this means that, if the kernel was compromised, the attestation process could also not even start al all.

The solution, among the one presented, adopted for this project is the second one, that avoids the complexity of the first one but still conveys a good security level, performing *secure boot* and *authenticated boot*.
Once the boot process and the kernel version are verified, it is time to perform the *Continuous Remote Attestation Process* installing, configuring and bootstrapping Keylime. But let's first see how the Keylime framework was modified to be used in this project.

6.2 Architecture of the proposed solution

The solution proposed in this first phase of the project sees two different architectures: the simpler one is composed by two physical nodes, the first one is the remote node, the *Attester*, that hosts the Keylime component *Keylime_ Agent*, while the other node hosts three different components: the *Keylime_ Tenant*, the *Keylime_ Registrar* and the *Keylime_ Verifier*. So we have a daemon running on the node, called Attester, that speaks separately with the three daemons hosted by the same machine, has shown in figure 6.2.

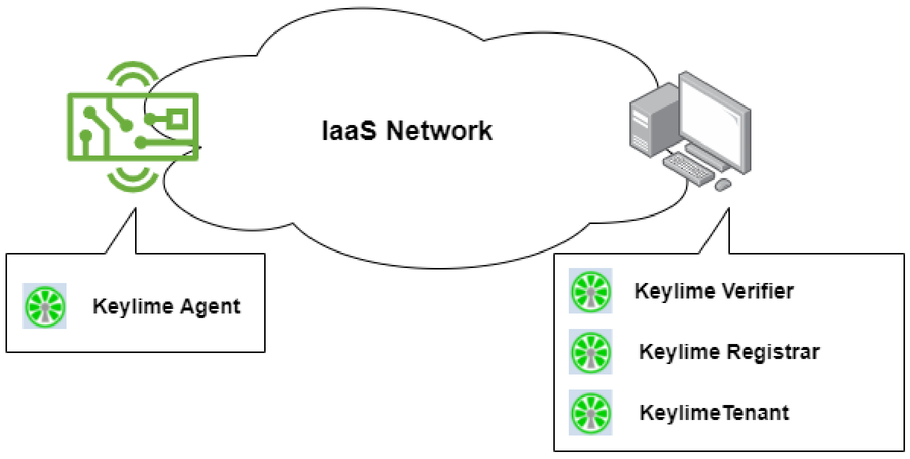


Figure 6.2. Architecture considering only two physical nodes

The second architecture is slightly different, since its structure contains four nodes: one of them is the same as before, and is the one hosting the *Keylime_ Tenant*, the *Keylime_ Registrar* and the *Keylime_ Verifier*, the other three components are virtual machines hosting the *Keylime_ Agent* component. It is important to point out that the three virtual nodes we are considering here are virtual machine created with the *Linux Kernel Virtual Machine, (KVM)*. Thanks to KVM, it is possible to create virtual machines and attach to them different hardware elements: among

these elements it is possible to find the TPM 2.0. Unfortunately, it is not possible to connect the physical TPM of the computer running KVM to the VM, but another solution is possible: using a **software TPM Emulator**. In fact by installing on the host machine, running KVM, a software TMP emulator it is possible to attach it to the virtual machine. In this way the VMs see the TPM as a real physical device, but what we are actually using is a software emulator.
This second architecture was built in order to simulate, albeit to a lesser extent, a cloud environment containing more than one node to be attested at the same time. This architecture is shown in Figure 6.3.

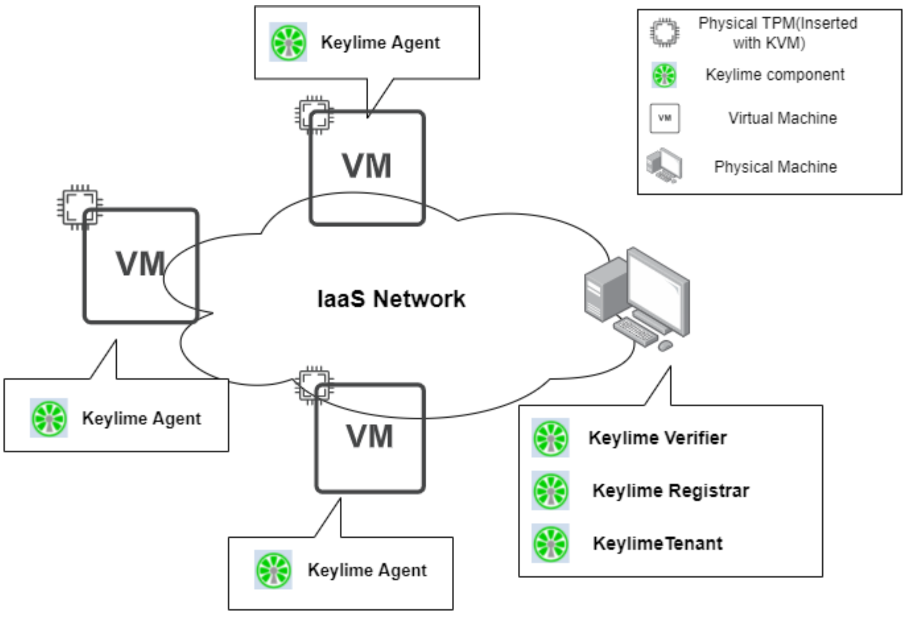


Figure 6.3. Architecture considering fours nodes, one physical and three virtualized

Thanks to the software TPM emulator installed on the host machine running KVM, both virtualized hosts and the physical one are seen by the *Tenant-Registrar-Verifier node* as physical nodes. The only real difference between these two kind of nodes is the TPM manufacturer certificate, that is used during the *Three Party Bootstrap Derivation Key Protocol* by the Tenant daemon. As specified in section 5.3, during the *B phase* of this Three Party protocol, the Tenant checks that the TPM credentials saved at the Registrar are valid: in particular it checks that the issuer of the EK_{cert} is a trusted TPM manufacturer. Here a problem arises: while the physical node has on board a physical TPM with an EK_{cert} issued by a trusted TPM manufacturer, the EK_{cert} of TPM of the VM was issued by the software emulator developer and cannot therefore be considered as *trusted* by the Tenant.
This is the real difference between the two nodes and that's the reason way we have two different architectures: in the second one, thanks to *Keylime configuration file*, the check performed in the B phase of the Key derivation Protocol on the EK_{cert} is skipped. Of course this check is of fundamental importance, since it validates the TPM credentials of the node we want to attest and skipping it is really dangerous, but this architecture's main goal is to test Keylime's Verifier when is has to monitor more than one Agent at the time.

6.3 Modifications made on the Keylime Framework

The **Keylime Framework** is an open source project and its code is freely downloadable from the official *GitHub repository*. Considering the two different architectures described above, it is possible to identify two categories of devices: the first one includes the device hosting the three components at the same time, while the second one includes both the physical node to be attested and the VMs.
Since this experimentation was born in the context of the ROOT project, where we have a network containing several kind of devices, among which there are also *small IoT devices* with limited resources in terms of storage and computing power, the devices chosen for the installation and testing of the various Keylime components were modelled according to the needs of the project itself.
The device used to install the *Keylime_ Tenant*, the *Keylime_ Registrar* and the *Keylime_ Verifier*, is a common laptop running an *Ubuntu 20.04* as OS. Since we are talking about a device that doesn't have particular limitations in terms of storage or performances, it can perfectly run the daemons contained in the repository as they are, without any modification. For this portion of the framework just some minor modifications were made.
The devices used to host the *Keylime_ Agent* were the ones shaped considering the IoT devices of the ROOT network: on the one hand we have the physical prototype described above and on the other hand we have the VMs, running an Ubuntu 20.04 Server Version as OS, with 15GB of storage and a 2 GB RAM. Since these devices have limited resources, even if they could have run the Keylime Agent properly as it is, some modifications were made in the code in order to make it lighter.
First of all, in order to save space on the disk, just the code and the libraries related to the Keylime_Agent were considered: this means that the Attester is not installing the whole Keylime Framework, but it is just downloading a repository containing a python script implementing the Keylime Agent. The code for this simplified version of the Agent is available at my personal *GitHub repository*.
The differences between this simplified version and the one available at the official Keylime GitHub repository are listed here:

- The biggest difference is the support for the **revocation framework**: even if Keylime provides a revocation framework, in this simplified version of the *Keylime_ Agent* this service in not supported. This decision was made because this was not the main focus of the work, since the first objective was to test if was possible to run the remote attestation process on devices like the one I worked on. This modification involves the *main()* function in the *Trust_ Agent/New_ Agent2510/agent.py* file;

- Another difference is that the support for the TPM emulator was removed. As already specified the goal of this first part of the experimentation was to work with devices with a physical TPM install. This modification involves the *main()* function in the *Trust_ Agent/New_ Agent2510/agent.py* file;

- Another difference is in the management of the UUID: by setting the related fields in the configuration file(*agent_ uuid*), Keylime provides several options to set or derive the UUID, (e.g it can be generated randomly, it can be evaluated as the hash of the EK or can be derived from the metadata service). This options are not supported by the simplified version of Agent, which only accepts that the UUID is directly assigned to the Agent. As for the previous two, also this modification involves the *main()* function in the *Trust_Agent/New_Agent2510/agent.py* file;

- The last modification was made in order to be able to perform some time measurements: the idea is to measure the time needed by the Agent to create the Quote. In the *do_GET* function in the *Trust_Agent/New_Agent2510/agent.py* file, some lines of code where added in order to read the time (from a timer) just before the quote is evaluated and then to read the time again, just after the quote is calculated. Then the difference between these two time readings is written into a file containing all the measurements related to the quote times,and it is available available at the path *var/lib/keylime/quote_creation_times*.

```
start_creating_quote = timer()
quote = tpm_instance.create_quote(nonce,
    self.server.rsapublickey_exportable, pcrmask, hash_alg)
imaMask = pcrmask
end_creating_quote = timer()
with open("quote_creation_times.txt", "a") as f:
   f.write("%f\n" % (end_creating_quote - start_creating_quote))
logger.info('Quote creation time: %f sec' % (end_creating_quote -
    start_creating_quote))
```

This repository also contains a python script, called *RAM_CPU_usage.py*, that, once run, starts to periodically check the usage of both the RAM and the CPU, in order to verify how many resources the Agent daemon is consuming. The results of this analysis are then written into a file available in the same directory where the script was run.

For what concerns the other three component of the framework, a slightly modified version of the Keylime 6.2.0 version is available at my personal *GitHub repository*. No modifications were made to either the Tenant or the Registrar.
With regard to the Keylime_Verifier the only modification made to the code is once again related to the need of making some time measurements. The goal of these measurements is to evaluate the time that passes from the Quote request made by the Verifier to the moment in which the Quote is verified and the *trust decision* is made. To do so, the same solution used for the Agent is followed: the time is read just before the GET request in made and it is read again once the quote is checked. The difference of these two times is then written into the *IR_attestation_times.txt* file , that will be created in the same directory where the Verifier daemon was run.

```
 start_IR_attestation = timer()

 res = tornado_requests.request("GET",
     "http://%s:%d/v%s/quotes/integrity?nonce=%s&mask=%s&vmask=%
s&partial=%s&ima_ml_entry=%d" % (agent['ip'], agent['port'],
    version, params["nonce"], params["mask"], params['vmask'],
partial_req, params['ima_ml_entry']), context=None)
```

```
response = await res
...
\\Some checks are performed on the response
...
failure = cloud_verifier_common.process_quote_response(agent,
    json_response['results'], agentAttestState)
if not failure:
    end_IR_attestation = timer()
    with open("IR_attestation_times.txt", "a") as f:
        f.write("%f\n" % (end_IR_attestation - start_IR_attestation))
```

These are all the modification that were made to the framework, then the rest of the operations were made by setting properly the parameters in the *keylime.conf* file. This is the configuration file of the framework and is divided in several sections, one for each of the components of the framework, plus a general section.

The installation steps and the commands needed to start properly all the components are available in the Appendix B. This is supposed to be a practical guide that leads the reader to a working environment where the Keylime framework, or at least its modified version, can be tested.

In the following, a different scenario, and the related modifications to the framework, will be presented: we are going to consider devices without a physical TPM installed on them, trying to understand which are the security implications that this condition have, and how to mitigate them modifying the framework.

Chapter 7

Keylime Continuous Remote Attestation of a physical node with a TPM Simulator

The second scenario in which we want to deploy Keylime, is that of a network containing only *nodes without a physical TPM installed*: if the previous chapter studied an environment involving only devices with a physical TPM, this one wants to describe the problems that have to be faced in a scenario including only devices without a physical TPM and the way in which these problems can be solved. The idea is to reach a solution in which both the kind of devices, with and without a physical TPM on board, can be remotely attested at the same time in the same infrastructure (that it the goal of Chapter 8). To do so, this two scenarios have to be described separately, outlining and solving the related problems, in order to be able to better understand the different criticality.

7.1 Devices involved in the analysis

The network at the centre of the ROOT project is based on simple devices and the study presented in this thesis aims to be as close as possible to the needs and constraints within the project. Therefore, just as the previous chapter considered both low-power nodes, like the prototype described, and virtual machines with limited memory and low performances, in this chapter we want to present an architecture made of devices with other kind of constrains and characteristics, trying to meet the requirements for the ROOT distribution of time network. These are much simpler devices than the one already presented, so simple that they do not even have a TPM on board. While TPMs are very common in more sophisticated devices such as laptops, it is very likely that one will find IoT devices that do not have this module on board. Along with the lack of a physical chip, another component that is most likely missing from these devices is the *CRTM*, already described in section 2.2.3, which is the first link of the Chain of Trust, ensuring that the expected bootloader is executed.
Overcoming the assumptions and consequent limitations made in the previous chapter, we are now going to consider nodes that do not have the CRTM and which,

instead of a physical TPM, have a **TPM Simulator installed,** a software solution emulating the functionalities of the chip. This software-based solution is a way to overcome the lack of hardware: obviously, this allows the device to benefit from most of chip's features, but the security properties of the device itself suffer.

In summary, the assumptions made in this chapter and the main characteristics of the devices that will be analysed are:

- All devices considered in the analysis *do not contain a CRTM or a physical TPM*, but a TPM Emulator is installed on them.
- Since we have to consider devices that should be as simple as possible, in order to cover even the worst cases, only devices on which the *TPM Access Broker* (TAB) and the *Resource Manager* (RM) are not installed will be analysed.
- The *IMA kernel module is enabled* on every device. This creates some problems because IMA expects to interact with a physical TPM: it is therefore necessary to implement some changes to overcome this conflict.

7.1.1 TPM2.0 Emulator and absence of the CRTM

The first assumption made is about the TPM2.0 Emulator and the CRTM: replacing the physical TPM with an emulator and not having the CRTM available means that several considerations must be made with regard to the security implications.

First of all the absence of the CRTM, either burned into the motherboard or written inside the BIOS, implies that there is no guarantee about which bootloader is executed: as it was already underlined, the CRTM is an immutable set of command which ensures that the right bootloader is loaded and run.
Secondly, the fact that the TPM is a software installed in *user-space*, implies that it can be run only once the bootloader is executed, the kernel is launched and the OS starts. This makes impossible for the TPM to perform both the *secure boot* and the *authenticated boot* phases. Since the TPM starts only after the bootloader is executed, it is not possible for IMA to extend the PCRs of the chip with the measurements it has taken of all the component executed before the Operative System starts. As it was already explained in section 6.1.1, these two phases, both the secure boot and authenticated boot, are very important from the point of view of the attestation process: they ensure that the right bootloader is executed, that the kernel is measured and this measurement is used to extend the expected PCRs and that the kernel is loaded correctly. Because of the lack of a physical chip and of the CRTM, the device owner cannot be sure of which version of the bootloader was executed, and consequently also which version of the kernel was run. For this reason, the first entry of the ML (Measurement Log) that is the *boot_ aggregate* (see section 3.2.2) will be set to all zeros, since IMA it is not able to extend the PCRs and this aggregate value is evaluated using the content of PCRs from 0 to 7.

Since we are talking about a problem that arises before the kernel itself is run, it cannot be solved without hardware solutions, but none of them will be proposed in the following: this means that the nodes inside the architecture being presented in this chapter do not implement any kind of secure boot or authenticated boot. The attestation process is therefore incomplete and only deals with the dynamic part, measuring every user-space applications and files.
The software emulator that was chosen for the implementation is the *IBM's Software TPM 2.0*, which refers to the TCG TPM 2.0 specifications: this software is an improvement of an implementation based on the source code donated by Microsoft.

7.1.2 TABRMD : Access Broker and Resource Manager Daemon

As already described in section 2.2.2, two important components of the TCG Software Stack (TSS) are the *TPM Access Broker* (TAB) and the *Resource Manager* (RM): the former manages concurrent accesses to the TPM in multi-process environments, guaranteeing that any TPM2 Tools operation performed for one process is not interrupted in order to serve another process; the latter manages the small amount on memory available on the chip, swapping in and out the context and other data related to the process that has to be served. They put themselves between the System API and the Device Driver (that could be an actual driver or a simulator) managing the access to the TPM .[27]
The *TPM2 Access Broker and Resource Manager Daemon* (TABRMD) is a daemon implementing this two components, remaining faithful to the specifications dictated by the TCG. The communication between daemon and clients is performed using the DBus and Unix pipes, in order to send and receive commands and responses but also for session management.

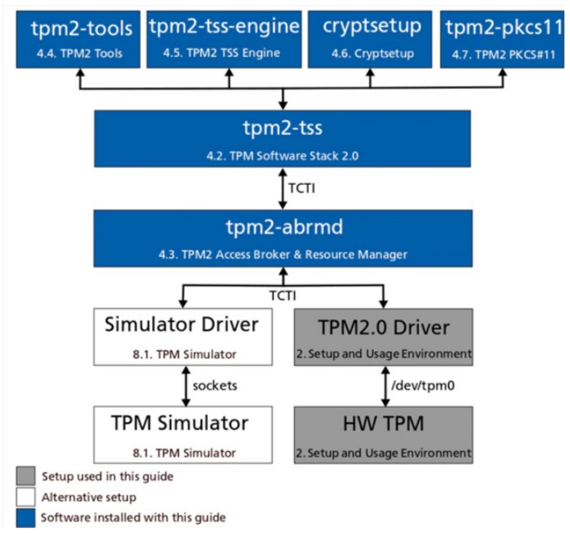


Figure 7.1. Structure of the TPM Software Stack and its components: role of the Access Broker and the Resource Manager

In order to properly install and run this daemon, some dependencies have to be met:

- GNU Autoconf
- GNU Autoconf archive
- GNU Automake
- GNU Libtool
- C compiler
- C Library Development Libraries and Header Files
- pkg-config
- glib and gio 2.0 libraries and development files
- libtss2-sys, libtss2-mu and TCTI libraries
- dbus

Among these requirements, it is possible to see that the TABRMD requires the *glib-2.0 package* and the *GObject* support for it: that is where the problem with this daemon arises. These two packages are not supported on every kind of GNU/Linux distro, especially on lighter versions of Linux, like the ones we should focus our attention on, since, as already pointed out, the devices taken into account by the ROOT project could likely be simple IoT devices. Therefore, for reasons of compatibility and support, in this part of the thesis only devices that are not compatible with these two packages will be considered: this means that, from now on, the devices taken into account *won't run the TABRMD*.
Neither of the two components will therefore be considered in the solution proposed in the chapter and alternative solutions will be found to make up for their absence. To solve this problem it is important to analyse the environment we are now considering: the devices chosen for this scenario are low performance machines, with few programs and daemons installed. Among them, the only one accessing the cryptographic module is the *Keylime agent daemon*, which uses it manly to perform the *Quote operation*. The TAB provides multi-user support for the TPM installed on the device, but since we are considering single-user devices with the Keylime agent being the only process accessing the chip, there is **no need for an Access Broker** to be installed on the devices making up this architecture.

For what concerns the role played by the **Resource Manager**, it cannot be just not considered as in the case of the TAB, because it has two important role: when a command is executed by the TPM, the RM has the duty to swap in the data (objects, context and sessions) related to the user that is about to be served, and to swap out the data currently stored in the TPM storage. Once the right data have been swapped into the TPM, the RM manages the limited amount of space available: it creates virtual handles for the context loaded in order to be transparent to the application that always use the same handle and monitors TPM

commands flushing session and restoring the context of previous connections. This first important task performed by the Resource Manager is not needed in our scenario: swapping in and out contexts related to different sessions and restoring these sessions is only needed if we put ourselves in a multi-user or in a multi-process environment. If we only have one user accessing the TPM and this is done through the operations performed by one single daemon (the Keylime agent), we don't need to cope with the management of the context and session swapping.
The RM, however, does not only manage contexts and sessions, it also manages *objects*: these are related to a single session and it is therefore necessary to find an alternative way to manage them, since a RM is not available. TPM objects could be either keys or data and are characterized by a public and perhaps a private part and they belong to one of the hierarchies. Each of the three persistent hierarchies the TPM has (Storage Hierarchy, Endorsement Hierarchy, Platform Hierarchy), is generated starting from a *seed*, which never leaves the TPM. This secret seed is then used as input of a **KDF** together with a public template which specifies the algorithm, the key size, the policy and the key type. The KDF is constructed to produce the same key if the seed and the input template are the same. The resulting primary key is then stored in TPM's volatile memory: it is therefore a *transient object*. Because of the limited amount of memory available inside the TPM, only few keys can be moved to the *persistent memory*. Thanks to the repeatable property of the KDF, even if the keys are flushed, they can be easily regenerated if the same input id provided (the seed is persistent, so cannot be lost).

In the solution adopted in this chapter, the management of the memory is performed in a very simple way, keeping into the transient memory only the strictly necessary data. The **EK** and the **AIK** are stored into the persistent memory, the NVRAM, surviving the power cycles. Each time the *Quote operation* is performed, the TPM creates an handle that acts as a reference to the position in which the AIK is stored: this reference is stored as a *transient object*. The transient (volatile) memory is limited, if it is not freed, managing properly the handles, it gets full and the Quote operations fails. If a Resource Manager is available into the platform, this problem is handled by it, but in the scenario we are now considering, this has to be done manually. A possible solution, albeit a quite invasive one, would be to modify the command for the quote operation provided by the *TPM Tools*, so that the command does not create a new handle each time it is invoked but allows the user to specify a previously used one.
The solution adopted here to overcome the absence of the Resource Manager is much simpler and radical: the only objects involved in the attestation process are the EK and the AIK, none of the transient object is used except the AIK key handle. So every time a new quote operation has to be performed, all the transient object are deleted with the *tpm2_flushcontext* command, in this way the memory is freed every time a new quote is sent and, when the *tpm2_quote* command is invoked again, the needed handle is recreated and the attestation process is performed properly.
It is really important to point out that this solution works in this very case because the TPM is used just to perform the Quote command that only requires the handle to the AIK stored in the NVRAM, and no other transient object is required.

7.1.3 IMA and TPM2.0 Emulator

We have already seen the Integrity Measurements Architecture in section 3.1: this kernel module is the component needed to perform what is called the *dynamic part* of the attestation process. It measures, following some user defined policy, executable, configuration and every other kind of file, keeping track of all these measurements in the *Measurement Log* (ML): a list of entries, where each entry contains the path of the accessed object and the related hash (its measurement). The ML is not the only way in which these measurements are recorded: each hash value is use to extend the PCR 10, which contains a sort of summary of all the files accessed. This module is essential to carry out the remote attestation process, that is why it is enabled and used on every node of the proposed architecture
Unfortunately, the integration of IMA with a software TPM is not so easy. The measurements kernel module expects to talk directly to a *physical chip* when it has to extend the measurement into the dedicated PCR (usually PCR 10). It actually works fine with an in-kernel TPM device emulator, the problem arises when a user-space TPM emulator is used. What is really hard to obtain in the latter case is to have the PCR 10 extended with the measurement of an application before the application starts: this is a crucial point in the attestation process since the measurement must be extended into the PCR before the application starts, in order to be sure that it was not modified in any way by the application itself. Since both the application to be measured and the TPM emulator run in the user-space, this cannot be guaranteed without an in depth modification of the IMA kernel module. For this reason IMA doesn't extend the PCR 10 of a TPM Emulator, which content is always equal to all zeros.
But let's make a step back to understand why it is so important for the attestation process that the measurements are extended into PCR 10.

As it was specified in section 5.3, in the description of the *Continuous Remote Attestation* process, the **Integrity Report** sent by the Attester contains the *quote* it has created and the *IMA ML*: the quote, which is signed, contains, among the other data, the value of the PCR 10 at the moment in which the quote is created. This value represents the result of the extension operation performed with each of the hashes contained in the ML: at the end, PCR 10 contains the aggregate of all the measurements recorded by IMA into the ML. By signing the quote, and thus the PCR 10 value contained in it, the Agent is indirectly signing the ML (by signing its aggregate containend in PCR 10).
When the Verifier receives the Integrity report and wants to attest its validity has to perform three actions:

1. *Verify the authenticity and freshness of the quote just received*: the Verifier checks that the signature evaluated over the quote was performed with the private part of the AIK registered to the Registrar and the quote has to contain the nonce sent by the Verifier, in order to avoid a reply attack. If these requirements are satisfied, the quote can be considered authentic and valid, as well as the value of the PCR 10 it contains;

2. *Verify the integrity of the ML*: starting from the first entry of the ML, using

the hash value contained in each line, the Verifier performs the extend operation until the value obtained matches the one reported into the quote (which it has just verified). If the values are the same, the integrity of the ML can be considered verified: in this way the ML could be considered *indirectly signed* and therefore can be used for the third step;

3. *Verify the validity of each file contained in the ML*: once the measurements inside the ML are considered trusted, this list can be used to check that each of the hash values it contains matches one of the values contained in the *whitelist* (the golden values).

If all these steps are successfully completed, the node can be considered trusted. But, if the value of the PCR 10 is not contained into the quote, or its value is not meaningful since IMA is not able to modify it, as it happens in the case we are considering where we have a TPM Emulator (the value of PCR 10 is equal to zero), ML's integrity cannot be verified and and the last two steps are compromised. What is needed to overcome the absence of communication between IMA and the TPM Emulator is to sign, directly this time, the ML that is sent into the Integrity Report and to have the Verifier validating this signature before going on with step 2: this is exactly what was done in the proposed solution.
The idea is to *sign the ML* with the same key that is used to sign the quote, the AIK, since the Verifier already knows its public part and send the signature into the Integrity Report, together with the quote and the ML. The Verifier then validates the signature over the ML just after it has verified the quote, so, if the verification is successful, it can proceed with the attestation of the device, reading the ML and checking that the hash of each file accessed is contained into the whitelist.

This kind of solution is not completely equivalent to the original one: the signature performed over the ML in user space guarantees that, if the the ML is modified while in the network, this modification is detected by the Verifier who is not able to verify the signature received in the IR. But if the attacker gains root access into the Attester, he can modify the file just before the signature is performed and this modification goes undetected.
With a physical TPM instead, even if the attacker gains access into the platform, he is not able to modify the value of the PCR 10 sent into the IR, because the only way in which PCRs can be modified is by an *extend* or a *reset* operation, but the properties of the hash algorithms guarantee that it is almost impossible for an attacker to perform a modification to a file and then perform the correct set of extension operations over a PCR, being able to obtain the desired hash value into the PCR.
The proposed solution is therefore less secure than the standard, but is still the best that can be done if no hardware solution is available and you don't want to modify the *TPM2_tools*.

7.2 Architecture of the solution

The architecture proposed in this second phase of the project, unlike that of the first phase, defines a single scenario: if in the first phase we considered both physical

nodes and virtual machines, in this part of the experimentation we are going to consider only **VMs** with the characteristics presented at the beginning of this chapter.
The architecture is made of four components, three of them are virtual machines hosting the *Keylime Agent daemon*, and, as in the previous case, the fourth machine runs at the same time the *Keylime Tenant daemon*, the *Keylime Verifier daemon* and the *Keylime Registrar daemon*. Also in this case, as well as in the second architecture of section 6.2, the nodes are virtual machines created with the *Linux Kernel Virtual Machine, (KVM)*: the difference is that this time no hardware TPM is attached to these VM. To overcome the absence of an hardware TPM, a software TPM is installed on these nodes, and the software chosen for this purpose is *IBM's Software TPM 2.0*. The VMs have the minimal version of the *Ubuntu 20.04 LTS* OS, they have *15G* of Storage and *2GB* of RAM.
The OS running on the physical node with the 3 components installed is the same as for the VMs, but this node is a common laptop with *256 GB* of storage and *8GB* of RAM, mounting an *Intel i7-7500U CPU*, with two cores and four logical processors.
All the nodes belong to the same private subnetwork, that is the local NAT created by KVM: the same result would have been obtained on a public network, modifying properly the *keylime.conf* file.

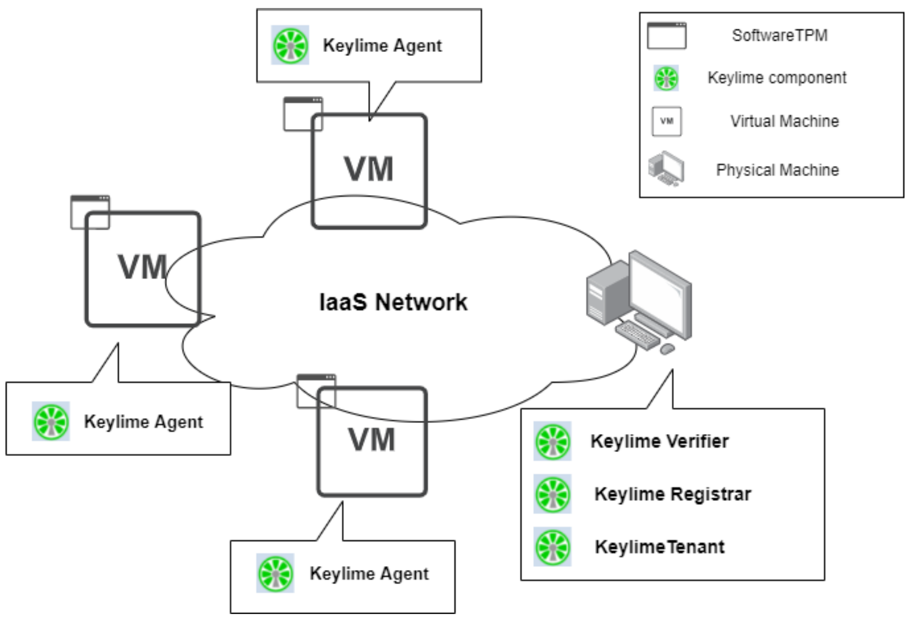


Figure 7.2. Architecture considering fours nodes, one physical and three virtualized with a TPM emulator installed

The daemons running in this architecture have some differences from the one of the previous chapter, since they have to be compliant with the limitations and assumptions made

7.3 Modifications made on the Keylime framework

As already analysed at the beginning of the chapter, some assumptions are needed in order to let this architecture work properly and in the way it is intended to work in the context of the ROOT project. Of course these assumptions lead to some characteristics that the nodes of the network should have, has already pointed out. In order to make the framework work properly in a scenario in which devices with these characteristics are inserted, and to make it compliant with the requirements of ROOT, some modifications to the code were needed. Each of the assumptions outlined requires modifications to the framework, be they small or a little more complicated:

1. The first assumption deals with the use of a TPM Emulator instead of a physical one. It is therefore necessary to find a way to to let the TPM Software Stack communicate with the software TPM installed on the platform: it is needed to configure the **Transmission Interface** (TCTI) specifying that the interface to be used in order to talk to the emulator, is the one linked to the TPM Emulator and not to the Access Broker. To do so, the */keylime/tpm/tpm_main.py* file was modified commenting line 43 and decommenting line 45:

   ```
   if 'TPM2TOOLS_TCTI' not in env:
          # Don't clobber existing setting (if present)
          #env['TPM2TOOLS_TCTI'] =
             'tabrmd:bus_name=com.intel.tss2.Tabrmd'
          # Other (not recommended) options are direct
             emulator and chardev communications:
          env['TPM2TOOLS_TCTI'] =
             'mssim:host=localhost,port=2321'
          # env['TPM2TOOLS_TCTI'] = 'device:/dev/tpm0'
   ```

   is it possible to see that several configurations are available: the default one is the *tabrmd*, that let the TCTI talk to the Access Broker, but the one chosen here is the *mssim* one, which redirects the TCTI to the software emulator, specifying the name of the host and the port that has to be contacted.

2. The second assumption deals with the absence of the TABRMD: the modification performed to implement the simple management of the memory, due to the lack of the RM, consists in flushing the transient objects every time a new quote is sent. To perform this operation, the *create_quote* function in the */keylime/tpm/tpm_main.py* file (line 988), has to be modified, adding the *tpm2_flushcontext* command:

   ```
   if self.tools_version == "3.2":
   ```

```
command = ["tpm2_quote", "-k", hex(keyhandle), "-L",
   "%s:%s" % (hash_alg, pcrlist), "-q", nonce, "-m",
   quotepath.name, "-s", sigpath.name, "-p",
   pcrpath.name, "-G", hash_alg, "-P", aik_pw]
elif self.tools_version in ["4.0", "4.2"]:
    command = ["tpm2_quote", "-c", keyhandle, "-l",
       "%s:%s" % (hash_alg, pcrlist), "-q", nonce, "-m",
       quotepath.name, "-s", sigpath.name, "-o",
       pcrpath.name, "-g", hash_alg, "-p", aik_pw]
self.__run(["tpm2_flushcontext", "-t"], lock=False)
retDict = self.__run(command, lock=False,
   outputpaths=[quotepath.name, sigpath.name,
   pcrpath.name])
```

Running the flushcontext command just before the quote command, allows the user to free the small memory of the TPM, avoiding that it gets full, blocking the TPM.

3. The last consideration done was about the impossibility for IMA to talk to the TPM Emulator, extending its PCRs: this means that the authentication and verification of the integrity of the ML cannot be performed by checking the value of the PCR 10, since it is meaningless, but the solution adopted here consists in signing the ML with the AIK. The modification needed to let the Agent perform the signature of the ML and for allowing the Verifier to skip the check and validate the signature were made on several files.
   The creation of the signature was implemented inside the */keylime/keylime_agent.py* file, at line 166:

```
if num_entries > 0:
   response['ima_measurement_list'] = ml
   response['ima_measurement_list_entry'] = nth_entry
   self.server.next_ima_ml_entry = num_entries
   signature = tpm_instance.ml_sign(ml,"signature_file")
   response['signature'] = base64.b64encode(signature)
```

   this modification involves a call to the *ml_sign* function that was created inside the */keylime/tpm/tpm_main.py* file. This function is the one that actually creates the signature over the ML using the AIK key handle, the password for accessing the AIK, passing these parameters, together with the signing algorithm and the format of the output file, to the TPM2_Tools command *tpm2_sign*:

```
def ml_sign(self, ml, signature):

    keyhandle = self.get_tpm_metadata('aik_handle')
    aik_pw = self.get_tpm_metadata('aik_pw')

    with open('msg','w') as f1:
        n = f1.write(ml)

    command = ["tpm2_sign", "-c", keyhandle, "-p",
        aik_pw, "-o", signature, "-f", "plain", "-g",
        "sha256", "msg"]
    retDict = self.__run(command, lock=False)

    with open(signature, 'rb') as f2:
        sign = f2.read()

    return sign
```

So at this point the signature is created (in the *ml_sign* function) and is sent into the quote response from the Attester (since it is added to the *response dictionary*). Once received by the Verifier, the signature has to be checked and verified: this is done in the same file as before, inside the function *_tpm2_checkquote* :

```
if ima_measurement_list is not None:
    ret = cryptodome.rsa_verify(cryptodome.rsa_import_pubkey(
            aikFromRegistrar), ima_measurement_list.encode(),
               signature)
    if not ret:
         logger.error("IMPOSSIBLE TO VERIFY THE SIGNATURE
              OVER THE MEASUREMENT LIST")
    return None, False
```

The only thing missing for the verification of the integrity of the ML, is that the Verifier has to skip the check performed on the value of the PCR 10: to do so the *found_pcr* variable inside the *_process_measurement_list* function in the */keylime/tpm/tpm_main.py* file is set to true and never modified. This variable is used inside the code as a flag: it is set to False at the beginning of the function and if, during the execution of the function, the value of the PCR 10 contained into the quote matches the value evaluated over the ML, the variable is set to True. By setting the variable directly to True, what we are indirectly doing is disregarding the result of the PCR 10 check, and therefore effectively skipping it.

These modifications make the framework meet all the requirements and the assumptions defined at the beginning of this chapter, making the framework suitable for the attestation of nodes which have a TPM simulator instead of a physical chip, which do not have a CRTM, providing a solution for attesting the integrity of the ML and managing the internal memory of the TPM.
Now that both the scenarios have been analysed individually, the next step would be to outline a solution for a scenario which considers all these different devices together.

Chapter 8

Keylime Continuous Remote Attestation of the ROOT infrastructure

In the last couple of chapters two different scenarios were presented: *Chapter 6* was dedicated to the analysis of the characteristics of a network where only devices with a physical TPM installed on them were considered, but also to the description of the way in which Keylime's code has been modified to address the requirements that a network like this has. *Chapter 7* described the modifications that the framework should undergo if the network in which it is used only contains devices without a physical TPM installed on them and that do not offer support for the TPM Access Broker and Resource Manager Daemon. These two environments are quite different as well as the patches the framework requires in the two different cases. The goal of this last chapter is to try to find a solution that leads to a scenario in which the framework could manage at the same time the two kind of devices: machines with a physical TPM on board and machines with a TPM software Emulator installed. A description of the way in which Keylime reacts to the discover of some potential attacks is then given, analysing some simple attacks and verifying that the framework reports them properly, recognizes the untrusted status of the compromised node and gives a brief description of what happened .
The last part of the chapter is focused on the performances of the framework in all the different scenario analysed, considering also the various devices involved in the whole project: the metrics that will be taken into account are the time needed for the attestation process and the resources needed, in term of computational power and storage.

8.1 Description of the proposed Architecture

The goal of the thesis is to obtain a framework capable of continuously attesting the status of the devices making up the network of an *IaaS*, which is based on a network that should be compliant with the requirements of the ROOT project. As already stated, the ROOT's time precision distribution network could be made of several nodes that can be ideally divided into 3 categories:

- *Tenant nodes*: this first category of nodes is the one that includes the devices running the three components that are usually managed by the Tenant, which are the *Keylime_ Verifier*, the *Keylime_ Registrar* and of course the *Keylime_ Tenant*. These components could be hosted either by different machines or by the same one and the devices running them have no special requirement or limitation in term of performances or resources, other than compatibility with all the packages needed for the installation of the framework;

- *Attester nodes with a physical TPM*: this category of nodes is the one including the devices already described in Chapter 6, that could either be VMs with the support that allows the user to let the VM consider the software TPM emulator running on the host machine as a physical component or low computational power nodes that mount a physical TPM on them. These nodes are the ones that have to be attested, running the *Keylime_ Agent* daemon, and are characterized by a small amount of memory and low performances;

- *Attester nodes without a physical TPM*: as the previous category of nodes, these machine are characterized by a small memory and a low computational power. The difference with the elements belonging to the previous group, is the fact that in this case no physical TPM is available and the cryptographic module is replaced by a software emulator; furthermore, in order to include even the simplest devices in this category, it was decided that all devices without a physical TPM would also be incompatible with the TABRMD.

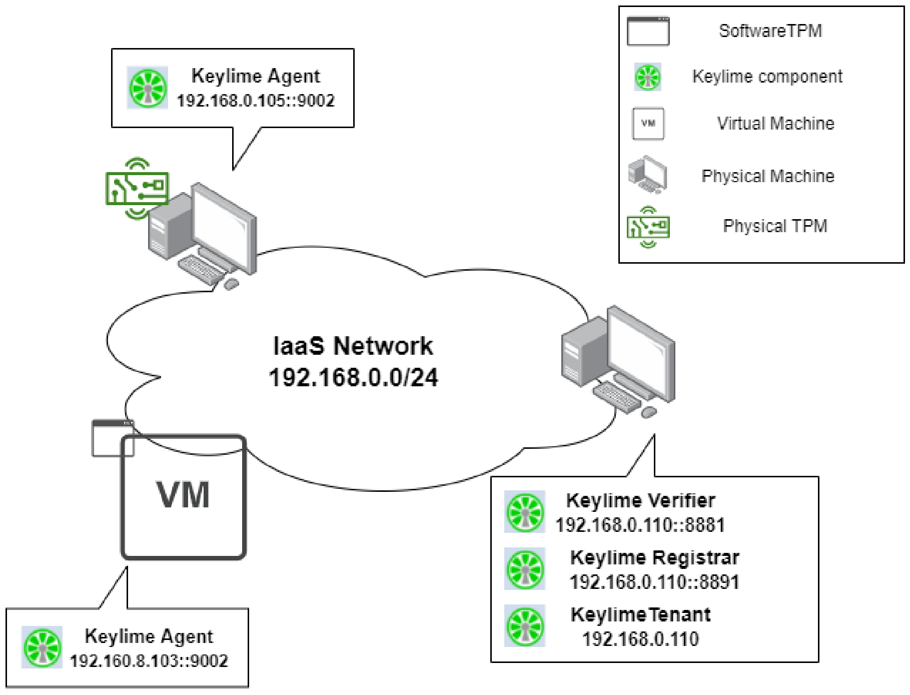


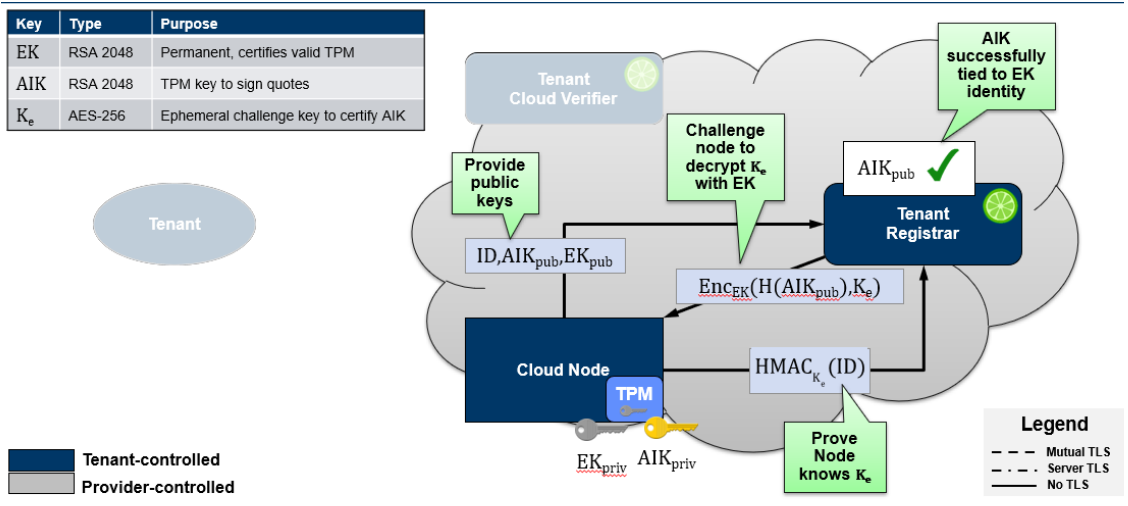
Figure 8.1. Proposed architecture for the unified Tenant

The architecture that is going to be presented in this section contains three nodes that were installed physically inside a laboratory and all plugged to the

same private network, one for each of the categories just defined: it was decided to host the three components under the Tenant control, the *Keylime_ Verifier*, the *Keylime_ Registrar* and the *Keylime_ Tenant*, on one single machine in order to be easily managed while using the framework. The machine chosen for this purpose is a HP laptop, mounting a Intel(R) Core(TM) i7-3520M processor, with 4 CPUs working at 2.90 GHz and 8 GB of RAM.
The node used to represent the second category is the already described prototype from Chapter 6, whose characteristics are available in this Appendix A.
Finally, the third node composing this architecture comes from the third group and has all the characteristics defined in the previous chapter (section 7.1): it is hosted by a Microsoft NUC mounting a Intel(R) Core(TM) i5-5300U processor, with 4 CPUs working at 2.30GHz and 16 GB of RAM.

8.2 Modifications made to the Keylime Framework

The main purpose is to create a sort of *"Unified Tenant"* capable of managing at the same time both devices which mount a physical TPM and devices which doesn't. In this way it is possible for a Tenant to perform the continuous remote attestation process simultaneously on both the kind of devices, obtaining a framework that is much closer to what the real structure of the ROOT architecture needs.



Key	Type	Purpose
EK	RSA 2048	Permanent, certifies valid TPM
AIK	RSA 2048	TPM key to sign quotes
K_e	AES-256	Ephemeral challenge key to certify AIK

Figure 8.2. Keylime Registration phase (Source [28])

The idea is to keep all the modifications made to the framework so far, adding a logic to define if the code that should be executed is the one related to a device with a physical TPM or to a device with a software TPM. The solution chosen to reach this goals consists in the addition of a field inside the configuration file, specifically inside the *cloud agent* section: the field is called **physical_tpm** and it could be equal to *True* or *False*, respectively if the agent does or doesn't have a physical TPM installed on it. What happens next, is that the Agent has to read that field and

add that information to the data structure sent top the Registrar in the registration phase. The registrar then inserts this information into its database, so that if the Tenant has to perform the attestation process, it can retrieve the information from the Registrar and perform the right set of commands, which depends on the value of this new filed.
In figure 8.2 it is possible to see the registration phase performed by the Agent: the modification proposed consists in sending, together with the information related to the ID, the AIK and the EK of the Agent, also the information about whether the node that is registering to the Registrar has or has not a physical TPM installed on it.

From the Agent point of view, a little modification was needed into the *keylime/keylime_agent.py* file, in the main function: the *physical_tpm* field is read from the configuration file and sent to the Registrar (which will store it into its database) thanks to the *doRegisterAgent* function:

```
physical_tpm = config.getboolean('cloud_agent', 'physical_tpm')

# register it and get back a blob
keyblob = registrar_client.doRegisterAgent (registrar_ip,
    registrar_port, agent_uuid, ek_tpm, ekcert, aik_tpm,
    contact_ip, contact_port, physical_tpm)
```

To have the Registrar save the value of the variable *physical_tpm* into the database, some slight modifications to the Registrar db and to the Verifier bd were needed, adding the field to the Agent table into the databases: this is done modifying the database files *keylime/db/registrar_db.py* and *keylime/db/verifier_db.py*.

It is important to remember that the validation of the EK certificate, performed by the Tenant in the Bootstrap phase, should be performed only if we are considering a physical TPM, since it is not possible to verify the EK certificate of a software TPM. Thus a check inside the *check_ek* function, into the *keylime/tenant.py* file, should be performed on the field considered, in order to skip the check on the certificate if it is not needed:

```
def check_ek(self, ekcert, physical_tpm = 1):

if physical_tpm is not None and physical_tpm == 0:
    logger.info("Skipping ekcert check due to the absence of a
        Physical TPM")
    return True
```

In the same way there is a difference on the validation of the ML authenticity if the platform contains a TPM chip or not: in the case the module is physically present, the PCR 10 value should be check, but if it is not, this check should be skipped and a signature over the ML should be evaluated, sent and then validated. This is done in two different files: the check on the PCR is performed, only if a physical TPM is present, into the *_process_measurement_list* function in the *keylime/ima.py* file, so a check is needed here:

```
if physical_tpm:
    found_pcr = (pcrval is None)
else:
    found_pcr = True
```

by setting the value of the *found_pcr* varaible (which is a flag set to true ongly if the check over the PCR 10 is successful) to True, we are skipping the check on the PCR.
The signature of the ML, whose implementation was described in Chapter 7, is performed into the *keylime_agent* file:

```
if not self.server.physical_tpm:
    signature = tpm_instance.ml_sign(ml,"signature_file")
    response['signature'] = base64.b64encode(signature)
```

Once performed by the Agent, the signature has to be validated, only if present, by the Tenant, and this is done in the *_tpm2_checkquote* function in the *keylime/tpm/tpm_main.py* file:

```
if ima_measurement_list is not None and physical_tpm == 0:
     ret = cryptodome.rsa_verify(cryptodome.rsa_import_pubkey
              (aikFromRegistrar), ima_measurement_list.encode(),
                 signature)
     if not ret:
         logger.error("IMPOSSIBLE TO VERIFY THE SIGNATURE OVER
             THE MEASUREMENT LIST")
         return None, False
```

The last modification that has to be made in order to obtain a Tenant capable of managing different devices at the same time, is in the *create_quote* function of the *keylime/tpm/tpm_main.py* file, deciding whether it is needed to flush the memory of the TPM or not, overcoming the eventual lack of space into the RAM:

```
command = ["tpm2_quote", "-c", keyhandle, "-l", "%s:%s" %
    (hash_alg, pcrlist), "-q", nonce, "-m", quotepath.name, "-s",
    sigpath.name, "-o", pcrpath.name, "-g", hash_alg, "-p",
    aik_pw]

if physical_tpm == 0:
    self.__run(["tpm2_flushcontext", "-t"], lock=False)
    retDict = self.__run(command, lock=False,
        outputpaths=[quotepath.name, sigpath.name, pcrpath.name])
```

The framework is ready to be installed except for one single modification that has to be performed manually just before its installation: the *_get_cmd_env* function contains the lines of code related to the configuration of the *Transmission Interface* (TCTI): if the Agent has a physical TPM installed, the *TPM2TOOLS_TCTI* environment variable should be set in order to have the TSS communicating with the TABRMD; if the Agent doesn't have a physical TPM, the variable should be set in order to have the TSS talking directly to the TPM Emulator (setting the *TPM2TOOLS_TCTI* environment to "mssim"). In the former case line 43 should be uncommented and line 45 commented, in the latter case the line to comment is the 43 and the one to keep is line 45.

```
def _get_cmd_env():
    env = os.environ.copy()
    lib_path = ""
    if 'LD_LIBRARY_PATH' in env:
        lib_path = env['LD_LIBRARY_PATH']
    if 'TPM2TOOLS_TCTI' not in env:
        # Don't clobber existing setting (if present)
        env['TPM2TOOLS_TCTI'] =
            'tabrmd:bus_name=com.intel.tss2.Tabrmd'
        # Other (not recommended) options are direct emulator and
            chardev communications:
        # env['TPM2TOOLS_TCTI'] = 'mssim:host=localhost,port=2321'
        # env['TPM2TOOLS_TCTI'] = 'device:/dev/tpm0'
    env['PATH'] = env['PATH'] + ":%s" % config.TPM_TOOLS_PATH
    env['LD_LIBRARY_PATH'] = lib_path + ":%s" %
        config.TPM_LIBS_PATH
    return env
```

The last set of modifications that has to be taken into account, are the once that were made with the goal of take some measurements to better understand the performances of the framework in terms of time needed for the attestation process and consumptions of RAM and CPU: this set of modification was already described in section 6.3.

8.3 Attacks to the testbed and response from the framework

The purpose of the Keylime framework is to attest the trusted status of a remote node: this means that all the files stored into the node should be known and measured and that every change is recorded and registered. This means that, whenever a file is added, modified, or accessed but this was not supposed to happen, the framework should report this event to the Tenant, that can react accordingly.
In order to test the correct functioning of the framework, it is useful to run some simple attacks emulating the behaviour that an attacker would have. The idea is to modify some file, or to create a new one and access it, and verify that the Verifier reports what happened, indicating which was the file or the files that didn't match

the golden values contained inside the whitelist.

The scenario that was taken into account during the experimentation is the one of the architecture just described in this chapter: it was assumed that an attacker was able to deploy a script to the target node, maybe using a vulnerability of one of the protocols used inside the network, or maybe it was physically installed on it, or it was present on the machine before the attestation process begun and remained silent for long time. The assumption is that, somehow, this script is stored in the machine, that the machine has **netcat** installed on it, which is plausible since it is installed on most of Linux Distros and that this scripts runs. The script just contains a bash command, the netcat command that allows a machine to connect to a remote node and execute a bin file, in this case the executable of the bash:

```
$ nc -e /bin/bash <IP_of_the_Attacker> <port_to_be_contacted>
```

This command let the machine that runs it to connect to the device whose IP address and port are specified. If this other device, let say the attacker, is listening on that very IP address and port by running the command:

```
$ nc -lnvp <port_to_expose> -s <IP_of_this_machine>
```

the attacker will see on its display the bash command line of the target machine: what happened is that the attacker was able to force the target machine to connect to itself, opening a **reverse shell** and gaining the access to that node.
Starting from this assumptions, the attacks that were run are two:

- **Creation of a new file**: we are now considering a situation in which the Keylime Agent is running on the machine, the node is registered to the Registrar and the Verifier is performing the continuous remote attestation process: at this point, a new file is created. Depending on where the file is created, two different reactions can be obtained: if the file is created and than saved inside a repository whose path is contained in the *excludelist* considered by the Tenant, the framework just keeps going with the attestation process since the Verifier doesn't find any integrity violation and this is right since that repository is not of interest to the Tenant. When instead the file is created and saved inside a repository that doesn't match any path in the *excludelist*, the Verifier finds the Integrity violation since the first quote sent by the Agent after the creation of the file will contain the measurement of the file itself. The problem is that the Verifier doesn't have any value related to this file in its whitelist, so the node is marked as untrusted. In a situation like this, what happens is that the Verifier notifies the user that the digest of this new file is not present in the whitelist, stopping the attestation process and scpecifying the path of the new file;

- **Modification of an executable file**: the situation we are considering is exactly the same as the situation above but this time we are considering

an existing file, that was therefore already measured, whose measurement is contained in the whitelist of the Verifier. The attack consist in modifying the current version of a daemon, in the case of the attack run is the *ptpd* file, adding a vulnerability to the daemon. The goal is to verify that, in the moment in which the new version of ptpd is executed, a new measurement is performed and a new Integrity report is sent to the Verifier through a Quote operation. Once the quote is received by the Verifier, it is checked and the measurement of the daemon will be missing in the whitelist: thus the attestation process is stopped and the path of the file that was modified is specified by the Verifier.
The strength of this system is that, thanks to the properties of the digest operation, it is barely impossible for the attacker to modify the executable in a meaningful way and obtain the same digest as before the modification.

These two simple attacks show the power of the Keylime framework: it is not able to prevent and stop an attack from happening, but it is able to detect that something is wrong, and to report it in few seconds.

8.4 Tests and performances

The last phase of this thesis work aims to evaluate the performances of the framework considering the testbed just described: the performances will be evaluated considering the consumption of CPU and RAM but also in terms of time required by the whole attestation process. The test that has been run is really simple: all the components where run and the attestation of both the node with a physical TPM and the node with a TPM emulator was started. The commands that were run are listed above, in the same order in which they were run:

- *Keylime Registrar*: on the node hosting the three Tenant components, the Registrar daemon was started. The idea is to run it using the command:

  ```
  $ sudo keylime_registrar
  ```

  in order to be able to see the log printed on the script and see what happens.

- *Keylime Verifier*: always using the command line opened on the Tenant node, the Verifier is started, by running the command:

  ```
  $ sudo keylime_verifier
  ```

- *Keylime Agent, no TPM*: on the node running the TPM emulator, a set of considerations has to be done. Here it is not enough to run the command that starts the agent, since we have to deal with the absence of the Resource Manager. Just before the command starting the daemon some actions on the TPM emulator need to be done: it is possible that the TPM that it is going to be used, was previously used for another attestation and stopped either because an error occurred during the attestation or because the process was manually interrupted. In order to bring the TPM back into a safe state we need to restart the emulator: once the emulator is rebooted, the TPM has to be started, and this is done by sending a startup command to the TPM and then, since a new attestation is about to begin, the TPM has to be cleared, deleting lockouts, endorsement and owner hierarchy authorization values. Once these actions are performed, the daemon can be run:

```
$ sudo systemctl restart tpm.service
$ tpm2_startup -c
$ tpm2_clear
$ sudo keylime_agent
```

- *Keylime Agent, physical TPM*: thi situation is completely different since there is no need to manage the absence of the RM or any other different situation. In this case it is just needed to run the daemon:

```
$ sudo keylime_agent
```

  At this point the two agent are registered to the Registrar, but the process is not started yet.

- *Keylime Tenant*: the attestation process will be started by the Tenant, which need to ask to the verifier to start attesting the remote node whose UUID and IP address are specified, together with the whitelist and the exclude list related to the agents.

```
#on the node with the TPM emulator)
$ sudo keylime_tenant -c add -v 192.168.0.110 -t
    192.168.0.103 -uuid emulator-node -f payload.txt --allow
    whitelist_emulator --exclude excludelist-emulator

#on the node with the Physical TPM
$ sudo keylime_tenant -c add -v 192.168.0.110 -t
    192.168.0.105 -uuid emulator_physical -f payload.txt
    --allow whitelist_physical --exclude excludelist_physical
```

Once all these commands are run, the attestation process of the the agents starts, and thank to the modification made to the code, described in section 6.3, the measurements of time and consumptions are taken.

8.4.1 Performance evaluation

The test just described is perfectly comparable to a normal use of the framework intended for the attestation of a remote node and has been conducted with the aim of carrying out some measurements on the performance. This evaluation is based on some metrics:

- *Quote Creation time*: the first metric used for the evaluation of the performance is the measurement of the time that the TPM, either physical of emulated, takes to create the quote to be sent to the Verifier. These measurements are taken thanks to the modification made on the code in the file *keylime/keylime_agent.py* (see section 6.3): the file *var/lib/keylime/quote_creation_times* is created, containing the time needed for the creation of every quote sent.
- *Attestation Cycle time*: this second metric is related to the whole attestation process, evaluating the time needed by the Verifier to request the quote, obtain the response from the Agent, verify the quote and take the trust decision. These measurements are conducted thanks to some minor modifications made on the *keylime/cloud_verifier_tornado.py* file (see section 6.3): the *IR_attestation_time.txt* file is created in the same directory in which the Keylime Verifier is run and contains the times needed to complete every attestation performed, from the moment in which the quote is requested until the moment in which the trust decision is made.
- *RAM and CPU consumption*: the last metric aims to monitor the use of resources, especially in terms of CPU and RAM usage. These measurements are performed by running the *RAM_CPU_usage.py* script, that creates a file containing the usage of the two hardware components.

```
CPU_usage = psutil.cpu_percent(interval=10)
RAM_usage = psutil.virtual_memory().percent
with open("CPU_RAM_usage.txt", "a") as f:
    f.write("CPU usage: %f\nRAM usage: %f\n" %
        (CPU_usage, RAM_usage))
```

In the following we are going to analyse the results obtained with respect to the three different metrics: all the measurements were carried out over a period of **2 hours**: the values obtained were then used to evaluate the average values that will be indicated in the following.

Quote creation times measurements

From the analysis on the data obtained measuring the quote creation times it was possible to see a substantial difference between the time needed by the hardware TPM and the time needed by the software TPM

It is possible to see how the time needed for the creation of the quote is much higher in the case of a software TPM: the average time value for the physical TPM

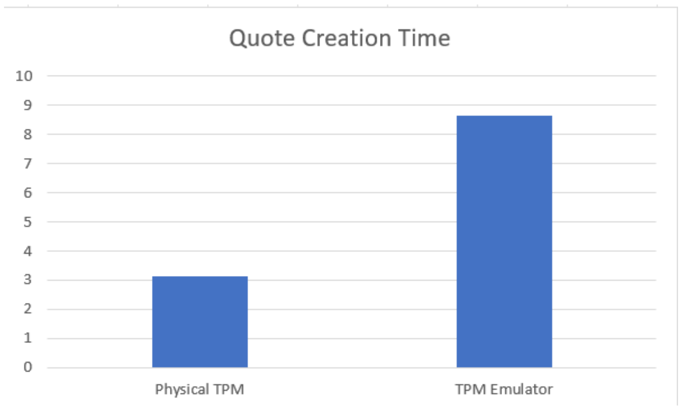


Figure 8.3. Comparison between the average quote creation time needed both by a physical TPM (value 3.14 s) and a software TPM (value 8.633 s)

is **3.143** seconds while a Software TPM takes on average **8.633** seconds.
This big difference is mainly due to the signature operation introduced in the case of a TPM emulator: to overcome the absence of communication between the emulator and IMA, a signature mechanism was adopted to certify the integrity of the ML, and this operation is the one requiring much more time.

Attestation process times measurements

The analysis of the time needed for the whole attestation process to start and to be completed highlights what was already visible in the previous analysis: the time needed for the attestation of the device with the emulator running is much higher then the time needed by the device with a physical TPM. Once again, this is due to the signature mechanism introduced in the emulator scenario: in this case what has to be considered is the time needed for the validation of the signature, evaluated over the ML.

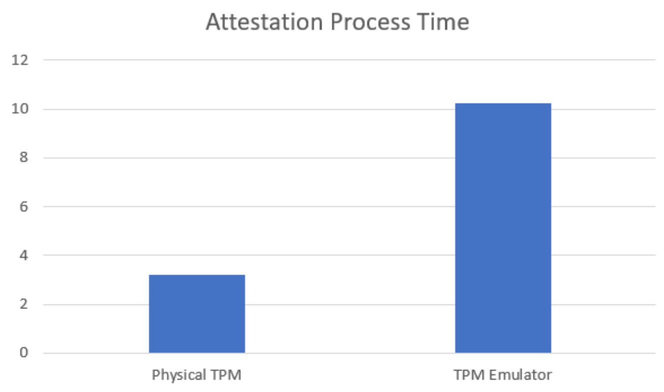


Figure 8.4. Comparison between the average attestation process time needed both by a physical TPM (value 3.22 s) and a software TPM (value 10.23 s)

By analysing the average time needed for the quote creation (**3,14 seconds**) and the corresponding value for the whole attestation process (**3,22 seconds**), in

the scenario of a physical TPM, is it possible to see that the difference between this two values (0,08 seconds), that is the time needed by the Verifier to validate the quote received, it is way smaller than the corresponding value in the scenario with a TPM emulator (**8.63 seconds** and **10.23 seconds**). This difference is due to the time needed for the verification of the signature in the second scenario.
The time needed for the attestation process to be completed is the sum of several time contributions:

- The time needed for the quote request to be sent by the Verifier to the node;
- The time needed for the Attester to create the quote: this time can contain also the time spent to evaluate the signature over the ML;
- The time needed by the Attester to read the ML;
- The time needed for the Attester to send the quote to the Verifier;
- The time needed by the Verifier to verify the quote response: this time could also consider the validation of the signature performed over the ML.

Consumptions of CPU and RAM

The last metric is the one analysing the consumptions in terms of CPU and RAM usage (in percentage), evaluated in 4 different situations: the scenario considering the TPM emulator but without the attestation process running, the same scenario but with the Keylime agent running, the scenario with the physical TPM and the keylime Agent daemon stopped and the case with a physical TPM and the Keylime Agent daemon running.

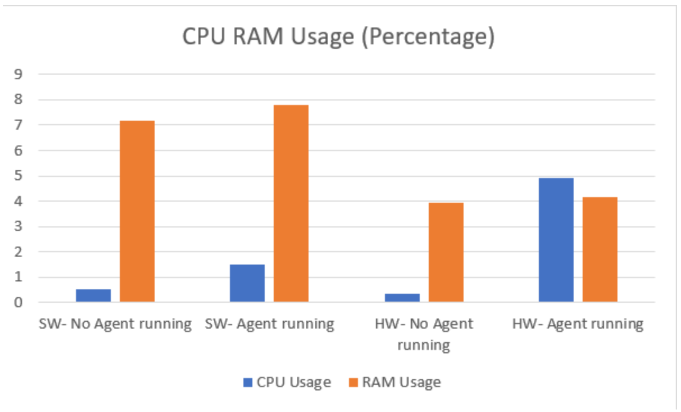


Figure 8.5. Comparison between the average percentage of use of CPU and RAM in both a physical TPM and a software TPM

The graph depicts the consumption penalty introduced by the running Keylime Agent: in the scenario considering a device with a TPM emulator installed on it, the increase of CPU usage caused by the running daemon is near to the 1% while the same increase in the scenario of a physical TPM is near to the 4%. This small difference is due to the type of the CPU mounted on the device with a physical

TPM, since is has much lower computational power than the one on the other device.
The consumption penalty, in terms of RAM usage, is instead quite similar in both scenarios: the increase of RAM consumption in percentage is near to the 0,5%.

The analysis just performed on the measurement carried out on this test, highlighted the great scalability of the framework, along with the capability of the Agent daemon to run on the device without placing a heavy burden on system resources, guaranteeing a low latency even if the attestation process is going on.

Chapter 9

Conclusions and future works

The main objective of this thesis was to define a solution that could fit within the ROOT project to better implement the remote attestation process. With the advent of 5G technology, accurate and secure time synchronization solutions are becoming increasingly important in the telecommunication field: although the requirements for synchronization have not become more stringent, the role of time synchronization has become much more critical.
The ROOT project is part of this reality and seeks to propose solutions that guarantee good synchronisation in time distribution networks. The new architecture proposed by the project, is based on GNSS receivers, ePRTCs clocks, C-GMCs and D-GMCs clock and dedicated protocols like the WB-PTP. These special elements and technologies are distributed over three different layers and combined with several kind of devices, commonly called management node: these particular devices could be the target of several cyber attacks and part of the ROOT resources are dedicated to the analysis, prevention and detection of these possible attacks.
The goal of this work is to enhance the ability of detect threats and attacks that the nodes making up the architecture may already have suffered: this is done by the implementation of a process of continuous remote attestation of all the nodes of the infrastructure. For this purpose, the CNCF open source project called Keylime Framework was chosen as the basis for the development of the attestation process. Keylime was considered to be perfectly suited to the reality just described, as it was developed for cloud environments and IaaS architecture, scenarios very close to the distributed architecture described in the ROOT project. Features like high scalability, making possible for a network manager to monitor a large number of devices without penalising the performance, the high compatibility with the major high level security systems and the high performance, especially in terms of latency introduced in the communications among nodes, make Keylime the perfect candidate for this work.

Once Keylime had been studied and analysed, it was possible to identify the features that needed to be modified in order to make the framework as close as possible to the needs of the ROOT project: the main requirement dictated by the ROOT project is the use, within the architecture, of very simple devices with little capacity in terms of memory and computational power. To overcome this lack of resources, in the first phase of the project a lighter version of the framework was

created to be installed on some of the devices composing the architecture: the experimentation was conducted considering a prototype based on a Rasbperry Pi 4 computer.
The second phase of the work was dedicated to the modification of the framework in order to make it compliant with the requirements of the simplest devices inserted into the architecture: the kind of devices analysed in this phase are low power machines without a physical TPM on board, whose role is covered by a TPM Emulator, and without the support for the daemon implementing the Access Broker and the Resource Manager. The absence of the AB was not a problem, since we are considering a single process scenario where there is only one daemon trying to access the TPM: the major problems were due to the absence of interaction between IMA and the software TPM and to the lack of a RM. The former problem has been overcome by introducing a signature mechanism over the ML sent inside the IR during the Quote operation: this is due to IMA's inability to write into PCR 10 of the TPM emulator and thus verify the integrity of the ML. The latter problem has been solved by introducing a simple management of the small memory available inside the TPM: the volatile memory is freed every time a new Quote operation is performed, in this way there is memory available for the creation of the needed objects.
The last phase of the thesis work consisted in making the framework capable of measuring both types of device, with and without a physical TPM, at the same time: for this purpose a field in the configuration file was added in order to have the framework act accordingly.

The project carried out and here described is just the starting point in the wider scenario of the ROOT project: this project is constantly evolving and its requirements evolve with it. Keylime also offers a revocation framework, which was not studied and used in this project: this feature of the CNCF framework is really useful since it allows to specify what to do if the attestation process going on with a node fails: it is possible to implement some revocation policies isolating the untrusted node or excluding it from the network.
Another aspect that could be very useful to improve is the management of the excludelist and the whitelist: by improving the whitelisting process, by implementing new policies, it would be possible to increase the performance of the framework, which would then manage fewer entries and therefore take much less time. The same effect could be obtained by improving the rules for writing the exclude list, avoiding the need to check all those files which may be considered useless for making the trust decision. The work carried out in the thesis, therefore, opens the perspective to multiple insights into the world of the remote attestation and especially in the use of the Keylime framework.

Bibliography

[1] The ROOT (Rolling Out OSNMA for the Secure Synchronisation of Telecom Networks) Project, https://www.gnss-root.eu/

[2] E. Falletti, D. Margaria, G. Marucco, B. Motella, M. Nicola and M. Pini, "Synchronization of Critical Infrastructures Dependent Upon GNSS: Current Vulnerabilities and Protection Provided by New Signals", IEEE Systems Journal, vol. 13, no. 3, pp. 2118-2129, Sept. 2019, DOI 10.1109/JSYST.2018.2883752

[3] DOD 5200.28-STD, "Department of Defense Trusted Computer System Evaluation Criteria", 1985, pp. 1-129, DOI 10.1007/978-1-349-12020-8_1

[4] Anthony Piltzecker, "Microsoft Vista for IT Security Professionals", 1st Edition, March 1, 2007, pp 123-193, ISBN: 9780080556147

[5] IEEE 1588-2019 - IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems, June 16, 2020, https://standards.ieee.org/standard/1588-2019.html

[6] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. V. Doom, “A practical guide to trusted computing“, IBM Press, 2007, ISBN: 978-0-13-239842-8

[7] R. Shirey, “Internet Security Glossary, Version 2.”, RFC-4949, August 2007, DOI 10.1109/MS.2010.160

[8] D. Mills, J. Martin, J. Burbank and W. Kasch, “Network Time Protocol Version 4: Protocol and Algorithms Specification”, RFC-5905, June 2010, DOI 10.17487/RFC5905

[9] V. Shankarkumar, L. Montini, T. Frost and G. Dowd, “ Precision Time Protocol Version 2 (PTPv2) Management Information Base”, RFC-8173, June 2017, DOI 10.17487/RFC8173

[10] W. Arthur and D. Challener, “A practical guide to tpm 2.0”, Apress Open, 2015, ISBN: 978-1-4302-6583-2

[11] Trusted Computing Group, ”Trusted Platform Module Library Part 1: Architecture”, TCG Published, November 8, 2019, https://trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-1-Architecture-01.16.pdf

[12] ”Trusted Computing Group TPM Main Part 1 Design Principles”, TCG Published, March 1, 2011, https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-1-Design-Principles_v1.2_rev116_01032011.pdf

[13] Trusted Computing Group, “TCG TSS 2.0 Overview and Common Structures Specification”, TCG Published, March 2, 2019, https://trustedcomputinggroup.org/wp-content/uploads/TCG-TSS-2.0-Overview-and-Common-Structures-Specification-Version-0.90-Revision-02.pdf

[14] TCG TSS 2.0 Enhanced System API (ESAPI) Specification, `https://trustedcomputinggroup.org/wp-content/uploads/TSS_ESAPI_v1p0_r08_pub.pdf/`

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

[16] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a tcg-based integrity measurement architecture", 13th USENIX Security Symposium (USENIX Security04), San Diego, CA, USA, August 9-13, 2004 `https://www.usenix.org/conference/13th-usenix-security-symposium/design-and-implementation-tcg-based-integrity-measurement`

[17] Remote Attestation tpm2-software community, `https://tpm2-software.github.io/tpm2-tss/getting-started/2019/12/18/Remote-Attestation.html`

[18] S. Ruffini, M. Johansson, B. Pohlman and M. Sandgren, "5G synchronization requirements and solutions", Ericsson Technology Review, `https://www.ericsson.com/en/reports-and-papers/ericsson-technology-review/articles/5g-synchronization-requirements-and-solutions`

[19] M. Pini, A. Minetto, A. Vesco, D. Berbecaru, LM. Contreras Murillo, P. Nemry, I. De Francesca, B. Rat and K. Callewaert, "Satellite-derived Time for Enhanced Telecom Networks Synchronization: the ROOT Project.", accepted for publication at IEEE MetroAeroSpace, 2021, Naples (Italy), June 23-25, 2021, DOI 10.3390/app11188288

[20] D. Margaria, G. Ramunno and A. Vesco, "Trust in GNSS-based Time Synchronization", LINKS Foundation, Torino, 2021

[21] M. Lipinski, T. Wlostowski, J. Serrano and P. Alvarez, "White Rabbit: a PTP application for robust sub-nanosecond synchronization", In 2011 IEEE International Symposium on Precision Clock Synchronization for Measurement,Control and Communication, pp. 25-30, DOI 10.1109/ISPCS.2011.6070148

[22] D. Berbecaru and A. Lioy, "Attack strategies and countermeasures in transport-based time synchronization solutions", 2021

[23] C. Guenther, "A survey of spoofing and countermeasures", Journal of the Institute of Navigation, 2014, DOI 10.1002/navi.65

[24] W. Alghamd and M. Shukat, "A detection model against precision time protocol attacks", 3rd International Conference on Computer Applications Information Security(ICCAIS), 2020, DOI 10.1186/s42400-021-00080-y

[25] C. DeCusatis, R. Lynch, W. luge, . Houston, P. Wojciak, and S.Gundert, "Impact of cyberattacks on precision time protocol", IEEE Transaction on Instrumentation and Measurement, vol. 69, no. 5, pp. 2172-2181, 2020, DOI 10.1109/TIM.2019.2918597

[26] 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, New York, NY, USA, December 2016, DOI 10.1145/2991079.2991104

[27] TCG Group, Family 2.0, "TCG TSS 2.0 TAB and Resource Manager Specification", Version 1.0, Revision 18, April 2019, `https://trustedcomputinggroup.org/wp-content/uploads/TSS_2p0_TAB_`

ResourceManager_v1p0_r18_04082019_pub.pdf

[28] Charles Munson, Nabil Schear and Martine Kalke, “Keylime: Enabling Trust in the Cloud”, Massachusetts Institute of Technology, 7 May 2019

Appendix A

Prototype characteristics

This Appendix contains the lists of the main characteristics of the physical prototype used to install the Keylime Agent and test the Kyelime framework.

Hardware characteristics

- Raspberry 4 Pi Model B - 4GB RAM;
- 16GB microSD card;
- Infineon Iridium TPM evaluation board (TPM 9670_Raspberry) with an OPTIGA SLI 9670AQ2.0 TPM (https://www.infineon.com/dgdl/Infineon-OPTIGA__SLx_9670_TPM_2.0_Pi_4-ApplicationNotes-v07_19-EN.pdf?fileId=5546d4626c1f3dc3016c3d19f43972eb).

Software installed and configured

- uBoot boot loader (version - 2020.04) with TPM enabled and boot script to wake up the TPM and to measure the boot components and extend some PCRs;
- Raspbian Buster Lite (kernel 4.19.118, no GUI)
  - kernel rebuilt with TPM2 driver built-in and IMA enabled with sha1
  - kernel with some driver patched to enable IMA after the TPM
  - sshd enabled on all network interfaces
  - default configuration: not stripped down not hardened (no firewall,
  - IPv4 and IPv6 enabled, no system proxy configuration)
  - WiFi not configured, but enabled
  - wired interface configured for DHCP
  - no proxy configuration
- TPM2 TSS v2.4.0

- TPM2 TABRM v2.3.1
- TPM2 Tools v4.2
- TPM2 TSS Engine v1.1.0
- Cryptsetup with TPM support v2.0.3

In the following a description of the available partitions id reported:

- /dev/mmblk0p1 => /boot (default)
- /dev/mmblk0p2 => / (default)
- /dev/mmblk0p3 => formatted ext4, not mounted, not used
- /dev/mmblk0p4 => /dev/mapper/EncSecPart => /mnt/encrypted

Appendix B

Keylime installation on a platform with a physical TPM installed

This Appendix's goal is to guide the reader during the installation of all the software needed in order to have a *Tenant* which can ask to a *Verifier*, relying on a *Registrar* to obtain TPM credentials, to perform the continuous remote attestation on one or more *Agents*. This guide refer to an Agent with a physical TPM installed on it.

B.1 Keylime Trust Agent Installation

This section aims to guide the reader towards the installation of the *Keylime_ Agent* on the remote node, either a physical one or a VM node. The Keylime Agent should be installed on the remote machine that is to be measured or provisioned with secrets stored within an encrypted payload delivered once trust is established. This guide refers to a physical node with a Physical TPM on board, or to a VM node which uses the TPM software emulator running on the host machine as a physical one.

This guide refers to a simulation executed on a machine running Ubuntu 20.4 server version, and is based on the Github repository *Linux TPM2 and TSS2 Software*, implementing APIs and infrastructure from the *TCG TSS2 specifications.*

This infrastructure is composed of:

- *tpm2-tss*: the TPM2 Software Stack proposed by the TCG gruop, which is composed by several layers allowing the user to have an easier interface to communicate with the TPM.
- *tpm2-abrmd*: a daemon implementing the TPM access broker and resource manager from the TCG.
- *tpm2-tools*: the source repository for the TPM2.0 tools based on the tpm2-tss which makes available the whole set of TPM's functionalities.

N.B. if you are using a virtual machine on KVM, remember to add a TPM in the hardware section of your virtual machine. In order to do this you have to install swtpm on your host machine. Use the CRB mode, selecting an emulated device for a TPM 2.0

B.1.1 Tpm2-tss

To build and install the tpm2-tss software the following software packages are required. In many cases dependencies are platform specific and so the following sections describe them for the supported platforms.

- GNU Autoconf
- GNU Autoconf Archive
- GNU Automake
- GNU Libtool
- C compiler
- C library development libraries and header files
- pkg-config
- doxygen
- OpenSSL development libraries and header files
- libcurl development libraries
- Access Control List utility (acl)

So the commands to be run in order to properly install the tpm2 software stack, on ubuntu 20.4, are:

```
$ sudo apt -y update
$ sudo apt -y install \
      autoconf-archive \
      libcmocka0 \
      libcmocka-dev\
      procps \
      iproute2 \
      build-essential \
      git \
      pkg-config \
      gcc \
      libtool \
      automake \
        libssl-dev \
        uthash-dev \
        autoconf \
        doxygen \
        libjson-c-dev \
        libini-config-dev \
        libcurl4-openssl-dev \
        acl \
        libglib2.0-dev

$ git clone https://github.com/tpm2-software/tpm2-tss.git tpm2-tss
$ cd tpm2-tss
$ sudo ./bootstrap
$ sudo ./configure
$ sudo make -j $(nproc)
$ sudo make install
```

B.1.2 Tpm2-abrmd

The tpm2-abrmd is a daemon, which is implemented using Glib, uses the DBus of the system bus and some pipes to communicate with the TPM.

Below the dependencies needed:

- GNU Autoconf
- GNU Autoconf archive
- GNU Automake
- GNU Libtool
- C compiler
- C Library Development Libraries
- pkg-config
- glib and gio 2.0 libraries

The daemon tpm2-abrmd can run as tss user or root. A good security practice is to run the daemon as unprivileged user, which requires creating a user account and group(in the following the name for the group and the user are "tss"). The account and associated group must be created before running the daemon as follow:

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

So the commands to be run in order to properly install the tpm2 access broker and resource manager deamon are:

```
$ git clone https://github.com/tpm2-software/tpm2-abrmd.git
$ cd tpm2-abrmd
$ ./bootstrap
$ ./configure --with-dbuspolicydir=/etc/dbus-1/system.d
$ sudo make
$ sudo make install
$ sudo ldconfig
```

B.1.3 Tpm2-tools

This repository is the one implementing all the high level cryptographic operations that can be performed by the TPM. The needed dependencied are:

- GNU Autoconf (version >= 2019.01.06)
- GNU Automake
- GNU Libtool
- pkg-config
- C compiler
- C Library Development Libraries
- ESAPI - TPM2.0 TSS ESAPI library
- OpenSSL libcrypto library
- Curl library

The libcurl dependency can be satisfied in many ways, and likely change with Ubuntu versions:

- libcurl4-openssl-dev 7.47.0-1ubuntu2.2
- libcurl4-nss-dev 7.47.0-1ubuntu2.2
- libcurl4-gnutls-dev 7.47.0-1ubuntu2.2

```
$ git clone https://github.com/tpm2-software/tpm2-tools
$ cd tpm2-tools
$ sudo ./bootstrap
$ sudo ./configure
$ sudo make -j $ (nproc)
$ sudo make install
```

N.B. At this point is possible to have some problem with the simulator on the host machine: in this case recompile and install again swtpm on the host machine

Once you have installed these 3 components reboot your system. After this operation the node is ready to install the Keylime Agent.

B.1.4 Installation of the Keylime Agent

The Keylime Agent that is about to be installed is a slight modification of the Keylime's version of the Agent. It it possible to get it from the GitHub repository *Keylime Agent* , and in order to get it the following commands are needed :

```
$ git clone https://github.com/Sync88/Trust_Agent.git
$ cd Trust_Agent
$ cd New_Agent2510
```

This commands will download the code but some python packages could be needed and they can be installed by running :

```
$ sudo apt -y update
$ sudo apt -y install python3-pip\
                python-yaml\
                python3-tornado\
                libssl-dev\
                swig\
                python3-dev\
                gcc\
                python3-gnupg\

$ sudo pip install pyyaml
$ sudo pip install simplejson
```

At this point the configuration of the network must be set up: the configuration file is called *keylime.conf* and you can find it in the repository just cloned but in order to have it read by the Trust Agent this should be copied into the */etc* folder. In order to have the framework working properly some fields should be changed:

in the [cloud_agent] section of keylime.conf

- *cloudagent_ip and agent_contact_ip*: it's the IP address the Tenant, the Verifier and the Registrar can contact to talk to the Agent (the second parameter is optional);
- *cloudagent_port and agent_contact_port*: it's the Agent's Port you want to expose to receive and send your messages to Tenant, Verifier and Registrar(the second parameter is optional);

- *registrar_ip*: it's the IP address the Agent can contact to register itself to the Registrar;
- *registrar_port*: it's the port the Registrar exposes and at which he can be reached;
- *agent_uuid*: it's the Agent identifier that is used to be enrolled at the Registrar and the Verifier;

Once all these fields are set, the Keylime Agent can be started by running the following command :

```
$sudo python3 ./agent.py
```

(it is possible to have a TABERROR after this command is run. In this case install *autopep8* running *sudo apt install python3-autopep8* and than the command *autopep8 -i agent.py*)

B.2 Installation of the Keylime Tenant, Keylime Verifier and Keylime Registrar framework on one single node

This section aims to guide the reader towards the installation of the three Keylime components on one single node: this guide refers a to computer running the Ubuntu 20.04 operating system. The repository that is about to be downloaded is a slight modification of the Keylime's version 6.2.0. It it possible to get it from the GitHub repository *Keylime*. This repository contains the whole framework by the components we are interested in are:

- *Verifier*: is the component responsible for asking the Agent for quotes and verifying them, reporting its trusted or untrusted state;
- *Registrar*: is the first component with which the Agent comes into contact and where it can register its TPM credentials. The Registrar verifies them and uses them for the remote attestation process;
- *Tenant*: it's a sort of wrapper that manages both the previous two components and allows the network administrator to register a new Agent to the Verifier and to start the continuous remote attestation process.

Also in this case, the *TPM2-tss*, *TPM2-abrmd* and the *TPM2-tools* packages have to be install, as shown above. In addition to them, also some dependencies should be respected:

```
$ sudo apt install libssl-dev swig python3-pip autoconf autoconf-archive \
libglib2.0-dev libtool pkg-config libjson-c-dev libcurl4-gnutls-dev
```

Once the dependencies are met, the code implementing the modified version of the Keylime framework could be downloaded and installed by running the following commands:

```
$ git clone https://github.com/Sync88/tenant_keylime_signature.git
$ cd keylime
$ sudo pip3 install . -r requirements.txt
```

In order to run the Keylime components, it is also needed to add TPM 2.0 Resource Manager user, configure the TPM TCTI and start the TPM 2.0 resource manager service as follows:

```
$ sudo useradd --system --user-group tss
$ export TPM2TOOLS_TCTI="tabrmd:bus_name=com.intel.tss2.Tabrmd"
$ sudo service tpm2-abrmd start
```

Once the code of the Keylime framework is downloaded and the access broker daemon started, the *keylime.conf* file has to be copied in the */etc* directory and then properly changed to configure the three components in the right way.

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

in the [cloud_verifier] section of keylime.conf

- *cloudverifier_ip* : it's the Verifier's IP address that will be the same as the IPs of Registrar and Tenant;
- *cloudagent_port* : it's the Verifier's Port you want to expose;
- *registrar_ip* : it's the Registrar's IP address used by the Verifier to ask to if the AIK of the Agent is trusted;
- *registrar_port*: it's the port the Registrar exposes and at which he can be reached;
- *quote_interval*: is the amount of time that should pass between an attestation and the other. If is set to 0, it will be done as fast as possible.

All the other parameters can be set with their default value.

in the [tenant] section of keylime.conf

- *cloudverifier_ip*: it's the IP address where the Tenant can find the Verifier;
- *cloudagent_port*: it's the port the Verifier exposes;
- *registrar_ip*: it's the IP address where the Tenant can find the Registrar;

- *registrar_port*: it's the port the Registrar exposes;
- *tpm_cert_store*: it's the repository where the Tenant keeps all the trusted certificate of the TPM manufacturer (this filed can be set with its own default value)
- *require_ek_cert*:this field enables the EK certificate check during the Bootstrap Key Derivation Protocol and is particularly important in our scenario. If we are considering the first architecture described in section 6.2 this field should be set to *True*. If we are considering the first architecture described in section 6.2 this field should be set to *False*.

In order to have the Tenant working properly and accept a new physical Agent, the certificate of the TPM manufacturer of the TMP mounted by the Agent, should be stored in the directory specified in the *tpm_cert_store* parameter. If we are considering VM nodes, the certificate could not be retrieved so it is necessary to set the *require_ek_cert* parameter to 'False'.

All the other parameters can be set with their default value.

in the [registrar] section of keylime.conf

- *registrar_ip*: it's the IP address you want to assign to the Registrar;
- *registrar_port*: it's the port the Registrar exposes;

All the other parameters can be set with their default value.

B.3 Launching Keylime and starting the attestation process

If all the steps described above are correctly executed and no problem has been found, we are ready to run all the components. The order in which the daemons are started is important too: specifically the Agent should be run after the Registrar and the Tenant should be run after the Verifier. So the correct order in which to start the daemons is :

1. The Registrar:

   ```
   $ sudo keylime_registrar
   ```

2. The Verifier:

   ```
   $ sudo keylime_verifier
   ```

3. (On the node hosting the Agent) The Agent:

   ```
   $ sudo python3 ./agent.py
   ```

Then, in order to have the Verifier starting the attestation process monitoring the Agent, we need to run the Tenant whit the following command:

```
$ sudo keylime_tenant -v <verifier_address> -t <agent_address>
 -f <file_to_encrypt_and_send> --uuid <uuid_of_agent>
 --allowlist <allowlist_file> --exclude <exclude_list>
```

this command is the most complicated one since it has some parameter:

- *verifier_ address*: is the IP address of the Verifier you want to contact (More than one Verifiers could be present in the framework);
- *agent_ address*: is the IP address of the Agent we want the Verifier to monitor;
- *file_ to_ encrypt_ and_ send*: this parameter is compulsory but the file included could also be empty, but could also contain some information the Tenant wants the Agent to receive;
- *uuid_ of_ agent*: this is the UUID of the Agent we want to monitor;
- *allowlist_file and exclude_ list*: these parameters are not mandatory and could also be set in the keylime.conf file. They represent the two files containing respectively the golden values of the file that should be measured and the list of files whose measurement should not be checked.

Once all of these commands are run, the Verifier will start monitoring the Agent the Tenant specified, and every time an integrity violation is detected, the Agent will be informed and will show an error message, and a similar error message will be shown by the verifier.

Appendix C

Keylime installation on a platform without a physical TPM and the abrmd installed

This Appendix's goal is to guide the reader during the installation of all the software needed in order to have a *Tenant* which can ask to a *Verifier*, relying on a *Registrar* to obtain TPM credentials, to perform the continuous remote attestation on one or more *Agents*. This guide refers to an Agent without a TPM installed on it and doesn't include the installation of the Access Broker and Resource Manager Daemon.

C.1 Keylime Trust Agent Installation

This section aims to guide the reader towards the installation of the *Keylime_ Agent* on the remote node, either a physical one or a VM node. The Keylime Agent should be installed on the remote machine that is to be measured or provisioned with secrets stored within an encrypted payload delivered once trust is established.

This guide refers to a simulation executed on a machine running Ubuntu 20.4 desktop version, and is based on the Github repository *Linux TPM2 and TSS2 Software*, implementing APIs and infrastructure from the *TCG TSS2 specifications.*

This infrastructure is composed of:

- *tpm2-tss*: the TPM2 Software Stack proposed by the TCG gruop, which is composed by several layers allowing the user to have an easier interface to communicate with the TPM.
- *tpm2-tools*: the source repository for the TPM2.0 tools based on the tpm2-tss which makes available the whole set of TPM's functionalities.
- *TPM 2.0 Simulator*: this is a software emulator of the TPM since a physical one is missing in this configuration.

N.B. Before starting the installation procedure it is necessary to uninstall libtss2 version 2.3.2 :

```
$ sudo apt remove libtss2-esys0
$ sudo apt autoclean && sudo apt autoremove
```

C.1.1 Tpm2-tss

To build and install the tpm2-tss software the following software packages are required. In many cases dependencies are platform specific and so the following sections describe them for the supported platforms.

- GNU Autoconf
- GNU Autoconf Archive
- GNU Automake
- GNU Libtool
- C compiler
- C library development libraries and header files
- pkg-config
- doxygen
- OpenSSL development libraries and header files
- libcurl development libraries
- Access Control List utility (acl)

So the commands to be run in order to properly install the tpm2 software stack, on ubuntu 20.4, are:

```
$ sudo apt -y update
$ sudo apt install libssl-dev swig python3-pip \
autoconf autoconf-archive git libglib2.0-dev libtool \
pkg-config libjson-c-dev libcurl4-gnutls-dev \
lcov pandoc liburiparser-dev libdbus-1-dev dbus-x11 \
automake gcc libgcrypt20-dev libcmocka-dev uthash-dev \
acl libssl-dev ssh

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

C.1.2 Tpm2-tools

This repository is the one implementing all the high level cryptographic operations that can be performed by the TPM. The needed dependencied are:

- GNU Autoconf (version >= 2019.01.06)
- GNU Automake
- GNU Libtool
- pkg-config
- C compiler
- C Library Development Libraries
- ESAPI - TPM2.0 TSS ESAPI library
- OpenSSL libcrypto library
- Curl library

The libcurl dependency can be satisfied in many ways, and likely change with Ubuntu versions:

- libcurl4-openssl-dev 7.47.0-1ubuntu2.2
- libcurl4-nss-dev 7.47.0-1ubuntu2.2
- libcurl4-gnutls-dev 7.47.0-1ubuntu2.2

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

C.1.3 TPM 2.0 Emulator

This section contains the instructions for the installation of the TPM 2.0 Emulator: this project is an implementation of the TCG TPM 2.0 specification. It is based on the TPM specification Parts 3 and 4 source code donated by Microsoft. The commands to download and install the emulator are the following:

```
$ wget https://jaist.dl.sourceforge.net/project/ibmswtpm2/ibmtpm1661.tar.gz
$ mkdir ibmtpm1661
$ cd ibmtpm1661
$ tar -xzvf ../ibmtpm1661.tar.gz
$ cd src/
$ sudo make
$ sudo cp tpm_server /usr/local/bin
```

In order to properly install and run the TPM daemon, some steps need to be done:

- Edit file /lib/systemd/system/tpm-server.service by adding the following content:

  ```
  $ sudo nano /lib/systemd/system/tpm-server.service

  [Unit]
  Description=TPM2.0 Simulator Server daemon
  Before=tpm2-abrmd.service
  [Service]
  ExecStart=/usr/local/bin/tpm_server
  Restart=always
  Environment=PATH=/usr/bin:/usr/local/bin
  [Install]
  WantedBy=multi-user.target
  ```

- Reload daemon, start the tpm-server.service service and check that its status is Active:

  ```
  $ sudo systemctl daemon-reload
  $ sudo systemctl enable tpm-server.service
  ```

- Configure TPM Command Transmission Interface (TCTI) for TPM 2.0 Simulator:

  ```
  $ export TPM2TOOLS_TCTI="mssim:host=localhost,port=2321"
  ```

- Start and check the status of the tpm simulatur service:

  ```
  $ sudo systemctl restart tpm-server.service
  $ sudo systemctl status tpm-server.service
  ```

Let's now check that the installation of the infrastructure is correct and that tpm2-tools are correctly configured with the TPM 2.0 emulator by reading the PCR banks (if the output contains the list of pcrs the output is the correct one) :

```
$ sudo tpm2_startup -c
$ sudo tpm2_pcrread
```

C.1.4 Enabling IMA

Before going on with the installation of the Keylime framework, it is necesary to enable IMA, adding kernel parameters and setting *ima policy* to measure only executable files:

- Edit the grub's file /etc/default/grub

  ```
  $ sudo nano /etc/default/grub
  ```

- In this file, modify to the GRUB_CMDLINE_LINUX kernel's boot parameter adding

  ```
  GRUB_CMDLINE_LINUX_DEFAULT="ima_tcb ima_hash=sha256"
  ```

  this parameter enables the measurement of all the files, of any kind, accessed by root.

- Update GRUB's configuration file to add the kernel boot parameter

  ```
  $ sudo update-grub
  ```

- Create directory /etc/ima

  ```
  $ sudo mkdir /etc/ima/
  ```

- Restart the system

Now it is possible to install the framework for the remote attestation. We have two different installation, one for the Agent and one for the Tenant Node (including the Tenant, the Verifier and the Registrar).

C.1.5 Installation of the Keylime Agent

The Keylime Agent that is about to be installed is a slight modification of the Keylime's version of the Agent. It it possible to get it from the GitHub repository *Keylime Agent* , and in order to get it the following commands are needed :

```
$ git clone https://github.com/Sync88/agent_keylime_signature.git
$ cd Trust_Agent
$ cd agent_keylime_signature
```

This commands will download the code but some python packages could be needed and they can be installed by running :

```
$ sudo apt -y update
$ sudo apt -y install python3-pip\
```

```
                python-yaml\
                python3-tornado\
                libssl-dev\
                swig\
                python3-dev\
                gcc\
                python3-gnupg\

$ sudo pip install pyyaml
$ sudo pip install simplejson
```

Once all the requirements are satisfied, it is possible to install the framework:

```
$ sudo cd keylime
$ sudo pip3 install . -r requirements.txt
```

At this point the configuration of the network must be set up: the configuration file is called *keylime.conf* and you can find it in the repository just cloned but in order to have it read by the Trust Agent, this file should be copied into the */etc* folder. (sudo cp keylime.conf /etc/). This file is divided in several sections, once for each role the framework is divided in. In the case of the Agent, we need to modify just the **[cloud_agent]** section, and in particular the following fields:

- *cloudagent_ip and agent_contact_ip*: it's the IP address the Tenant, the Verifier and the Registrar can contact to talk to the Agent (the second parameter is optional);
- *cloudagent_port and agent_contact_port*: it's the Agent's Port you want to expose to receive and send your messages to Tenant, Verifier and Registrar(the second parameter is optional);
- *registrar_ip*: it's the IP address the Agent can contact to register itself to the Registrar;
- *registrar_port*: it's the port the Registrar exposes and at which he can be reached;
- *agent_uuid*: it's the Agent identifier that is used to be enrolled at the Registrar and the Verifier;

Once all these fields are set, the *keylime_agent.service* has to be installed by running the *installer.sh* script that you can find in the Keylime repository. Remember to check the service's status (it should be "Active"):

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

If the state is "Failed", then try to clear the TPM and then start it again:

```
$ sudo tpm2_clear
$ sudo systemctl start keylime_agent.service
$ sudo systemctl status keylime_agent.service
```

At this point the installation on the Agent node is over. The only missing point is the creation of a whitelist that will be stored at the Verifier to validate the *TPM_quote*:

C.2 Installation of the Keylime Tenant, Keylime Verifier and Keylime Registrar framework on one single node

This section aims to guide the reader towards the installation of the three Keylime components on one single node: this guide refers a to computer running the Ubuntu 20.04 operating system. The repository that is about to be downloaded is a slight modification of the Keylime's version 6.2.0. It it possible to get it from the GitHub repository . This repository contains the whole framework, the components we are interested in are:

- *Verifier*: is the component responsible for asking the Agent for quotes and verifying them, reporting its trusted or untrusted state;
- *Registrar*: is the first component with which the Agent comes into contact and where it can register its TPM credentials. The Registrar verifies them and uses them for the remote attestation process;
- *Tenant*: it's a sort of wrapper that manages both the previous two components and allows the network administrator to register a new Agent to the Verifier and to start the continuous remote attestation process.

Also in this case, the *TPM2-tss* and the *TPM2-tools* packages have to be installed, as shown above. In addition to them, also some dependencies should be respected:

```
 $ sudo apt install libssl-dev swig python3-pip \
 autoconf autoconf-archive \
 libglib2.0-dev libtool pkg-config \
 libjson-c-dev libcurl4-gnutls-dev
```

Once the dependencies are met, the code implementing the modified version of the Keylime framework could be downloaded and installed by running the following commands:

```
$ git clone https://github.com/Sync88/tenant_keylime_signature.git
$ cd Trust_Agent
$ cd tenant_keylime_signature
$ sudo pip3 install . -r requirements.txt
```

Once the code of the Keylime framework is downloaded the *keylime.conf* file has to be copied in the */etc* directory and then properly changed to configure the three components in the right way.

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

The */etc/keylime.conf* file is divided in several sections, the one that have to be changed are:

in the [cloud_verifier] section of keylime.conf

- *cloudverifier_ip* : it's the Verifier's IP address that will be the same as the IPs of Registrar and Tenant;
- *cloudagent_port* : it's the Verifier's Port you want to expose;
- *registrar_ip* : it's the Registrar's IP address used by the Verifier to ask to if the AIK of the Agent is trusted;
- *registrar_port*: it's the port the Registrar exposes and at which he can be reached;
- *quote_interval*: is the amount of time that should pass between an attestation and the other. If is set to 0, it will be done as fast as possible.

All the other parameters can be set with their default value.

in the [tenant] section of keylime.conf

- *cloudverifier_ip*: it's the IP address where the Tenant can find the Verifier;
- *cloudagent_port*: it's the port the Verifier exposes;
- *registrar_ip*: it's the IP address where the Tenant can find the Registrar;
- *registrar_port*: it's the port the Registrar exposes;
- *tpm_cert_store*: it's the repository where the Tenant keeps all the trusted certificate of the TPM manufacturer (this filed can be set with its own default value)
- *require_ek_cert*: this field enables the EK certificate check during the Bootstrap Key Derivation Protocol and is particularly important in our scenario. In the scenario we are now considering, a platform without a physical TPM, this filed should be set to *False*, since we cannot obtain a valid ek certificate for a SW TPM.

All the other parameters can be set with their default value.

in the [registrar] section of keylime.conf

- *registrar_ip*: it's the IP address you want to assign to the Registrar;
- *registrar_port*: it's the port the Registrar exposes;

All the other parameters can be set with their default value.

In order to properly run the framework it is needed to delete some files related to the creation of the database: in particular the two files */var/lib/keylime/cv_data.sqlite* and */var/lib/keylime/reg_data.sqlite*, which are the file representing the databases of the CV and the Registrar. This operation is needed in order to recreate the databases with the first run of the framework.

Once the configuration file has been modified, run the *installer.sh* script in order to install the *keylime_verifier.service* and *keylime_registrar.service*. Let's start them and ensure that their status is "Active":

```
$ sudo services/installer.sh

$ sudo systemctl start keylime_registrar.service
$ sudo systemctl start keylime_verifier.service

$ sudo systemctl status keylime_registrar.service
$ sudo systemctl status keylime_verifier.service
```

If the state is "Failed", then try to clear the TPM and then start it again:

```
$ sudo tpm2_clear
$ sudo systemctl start keylime_registrar.service
$ sudo systemctl start keylime_verifier.service

$ sudo systemctl status keylime_registrar.service
$ sudo systemctl status keylime_verifier.service
```

In the same directory containing the whitelist, create 2 other files:

a. The exclude list that will be used to validate the quote, it contains the file whose measurements should be ignored.

b. The payload file that will be ciphered and sent to the agent. In this context in can be empty.

C.3 Launching Keylime and starting the attestation process

If all the steps described above are correctly executed and no problem has been found, we are ready to run all the components. The order in which the daemons are started is important too: specifically the Agent should be run after the Registrar and the Tenant should be run after the Verifier. So the correct order in which the services should be started is

1. The Registrar;
2. The Verifier;
3. The Agent.

Then, in order to have the Verifier starting the attestation process monitoring the Agent, we need to run the Tenant whit the following command:

```
$ sudo keylime_tenant -v <verifier_address> -t <agent_address>
 -f <file_to_encrypt_and_send> --uuid <uuid_of_agent>
 --allowlist <allowlist_file> --exclude <exclude_list>
```

this command is the most complicated one since it has some parameter:

- *verifier_address*: is the IP address of the Verifier you want to contact (More than one Verifiers could be present in the framework);
- *agent_address*: is the IP address of the Agent we want the Verifier to monitor;
- *file_to_encrypt_and_send*: this parameter is compulsory but the file included could also be empty, but could also contain some information the Tenant wants the Agent to receive;
- *uuid_of_agent*: this is the UUID of the Agent we want to monitor;
- *allowlist_file and exclude_list*: these parameters are not mandatory and could also be set in the keylime.conf file. They represent the two files containing respectively the golden values of the file that should be measured and the list of files whose measurement should not be checked.

Once all of these commands are run, the Verifier will start monitoring the Agent specified, and every time an integrity violation is detected, the Agent will be informed and will show an error message, and a similar error message will be shown by the verifier.

To see the log of each component could be seen by running:

a. On the Tenant node, to see the status of the attestation process, if the quotes are validated or if there are measurements that are not matched:

```
$ tail -f /var/log/keylime/cloudverifier.log
```

b. On the Tenant node, to see the status of the registration process at the Registrar:

```
$ tail -f /var/log/keylime/registrar.log
```

c. On the Tenant node, to see the status of the node:

```
$ sudo keylime_tenant -c status -u UUID1
```

d. On the Agent node, to see if the quote are sent to the verifier:

```
$ tail -f /var/log/keylime/cloudagent.log
```

Another possibility, if you want to see the logs directly on the bash without going into the log file, is to stop all the services (Registrar, Verifier and Agent) and running all the daemons by hand:

- The Registrar:

  ```
  $ sudo keylime_registrar
  ```

- The Verifier:

  ```
  $ sudo keylime_verifier
  ```

- (On the node hosting the Agent with the software Emulator) The Agent:

```
$ sudo systemctl restart tpm.service
$ tpm2_startup -c
$ tpm2_clear
$ sudo keylime_agent
```