POLITECNICO DI TORINO

Master degree course in Communications and Computer Networks Engineering

Master Degree Thesis

Hardware-Bound Virtual TPM for Cloud Computing Deep Attestation

Supervisors
prof. Antonio Lioy
dott. Marco de Benedictis

Candidate
Andrea Bertorello

July 2020

Acknowledgements

This thesis is the result of not only my work but also of all the others that were with me during this long path. I wish to express my deepest gratitude to my parents that believed always in me and that supported all my decisions whatever they have been. A special thanks also to my brother from which I am deeply connected and with whom I shared thoughts and difficult moments that did not concern only the course of study but also what is outside. Even if you are younger and it would be my job to give you the right advice you have often been wiser than me. Give it in and finish it too.

A special thanks to my companionship, Mauro, Vincenzo, Camillo and Giovanni, we suffered together, and we stressed each other but we had moments that we will always carry with us, starting from the discussions of music, football, politics and especially food. I also thank all the guys from Campus San Paolo, especially the sixth floor. You were the best, I spent fantastic moments with you. Thanks also to Salvo, Campus bartender and the only Sampdorian of the city, unfortunately most of the time we shared suffering but also hopes and dreams for our colors.

I can not fail to include in my thanks the Cisco CX Academy that selected me this summer for an adventure of a few months that turned into a real change of residence. I thank all my internship colleagues and program managers. I especially and infinitely thank my team, TAC Security AAA: Anton, Serhii, Eugene, Yevhen, Tasos, Anil, Olesya, Dana, Ilia, Uggen, Azad, Yazan and Piotr who cheer me up every day and make every day special. You have always helped me for anything both to grow professionally and in difficult times. If I managed to combine study and work, it is only thanks to you. Together is better.

The list is still long so I thank everyone I could not include, you are still in my memory.

Abstract

Nowadays the Cloud Computing paradigm changed the IT industry, reshaping the hardware provisioning and how services and infrastructures are developed. Cloud Computing is in fact a method to increase capabilities without the need for investment in infrastructure as well as in software. However, this evolution leads to integrity and security issues.

Data Integrity is nothing but the guarantee that the data is not accessed or modified by those that are not authorized. It can be achieved on a system through the usage of the Trusted Platform Module, through the collection and generation of integrity measures, it offers tamper resistance. Despite everything, this procedure cannot be supported in a virtual environment since a virtual TPM, *vTPM*, although it provides the same functionalities of a physical TPM ,*pTPM*, has the same weaknesses of any software.

Since Data Integrity is a crucial point in the Cloud Computing environment in order to provide reliability to the whole system, this thesis work proposes to investigate a solution for the Deep Attestation based on virtual TPM and its binding to a physical TPM, in order to retain the security strength of hardware-based root of trusts and the capability to correctly evaluate the reliability of a system.

Summary

In the last years, **Cloud Computing** has been considered as the new enabling paradigm able to solve computing expensive hardware investments and operational costs. Cloud Computing changed fundamentally the IT industry dramatically, reshaping the hardware provisioning and the infrastructure deployment. Its potential and efficiency made it become largely adopted in almost all the architectural choice. This upheaval paradigm , however, is not without a price. If on one hand, Cloud Computing has led the cloud consumer the possibility of being able to take advantage of unique properties such as elasticity, resiliency, fast provisioning and multitenancy. On the other hand, *security* and *privacy* were negatively affected leading them to be a critical aspect in the overall architecture. For this reason many solutions have been proposed and discussed, especially by NIST, CSA and in literature. Many critical issues can be addressed with traditional approaches. In other scenarios, however, it is necessary to adopt a cloud-specific approach.

Nevertheless, what it was mainly proposed were solutions based primarly on software implementations. Therefore, to rely on a software solution it is essential first of all be able to trust the system and the environment over which the solution is running. Trust in the execution environment is the critical element at the basis of all solutions. For this reason the **Trusted Computing Group** has developed a precise and detailed methodology to be able to obtain it. It is in this situation that the **Remote Attestation** is also involved. In theory, Remote Attestation is a method by which a *Prover* is able to authenticate the hardware and software configuration to a *Remote Verifier*. Based on measurements taken during the life cycle of the Prover, the Verifier is able to determine the level of trust in the platform integrity. It is based on two major components: the remote attestation protocol and the integrity measurements architecture. The leading actor in the Trusted Computing Group with its technology called **TC (Trusted Computing)**, based on is the key enabler **Trusted Platform Module**.

Trusted Platform Module (TPM) is an international standard for a secure cryptoprocessor, tamper resistance, designated to secure hardware which implements primitive cryptographic functions that are starting point of more complex features that are built on them. Despite the potential of the TPM, it goes in stark contrast to what is the paradigm of cloud computing. Among the many functionalities, the TPM has the task of saving within it aggregates of measures, which will then be used in the remote attestation process. The aggregate measurements are usually saved in special registers called **PCR**. Their primary use indeed provides a cryptographic record of the software state. The peculiarity of those registers is their update condition that is based on a one-way hash, which allows them not to be forged. They can be read to report the state of the machine and also signed for a more secure report. Here it comes the issue, a TPM implements only a strict and limited number of PCRs.

Virtual Machines in the cloud environment are of a considerable number, and this would lead to having to tie each virtual machine to a physical instance of a TPM. This fact, however, would clearly disagree with the scalability that has been achieved with the cloud computing paradigm and would also prevent the possibility of migrating the VMs being cryptographically linked to the physical TPM. For this reason, several approaches have been brought to the literature to overcome this problem. From this problem arises the virtual TPM instances. However, they have at their base what was the main problem to which they tried to solve or that they are purely software solutions, going to lose all the potential of a hardware solution.

For this reason, the goal work of the TPM 2.0 virtualization extension project is to design a strong binding between different *vTPM instances* and a single *pTPM* so that cloud users can benefit from an hardware bound vTPM identity, which results more secure compared to a standard vTPM software instance. Moreover, migration it can be now feasible, and in the Deep Attestation process, the chain of trust is finally extended to the hardware. Security and Privacy benefit from this implementation, in fact, vTPM is rooted in the physical Trusted Platform Module so that the physical platform itself protects its permanent state. Within this project, the thesis work focuses on the investigation of PCR binding for Deep Attestation use case. In other words, how to save the various PCR belonging to different virtual TPM within a single physical TPM instance.

The proposed solution was considered the best because despite the reproduction of the PCR inside the NVRAM can provide a classic Quote operation which is totally transparent to the Verifier or more in general to who uses it. Critical operations that concern the updating of PCR registers are managed by the physical TPM, which increases the general security of the implementation. What it has been obtained is. Therefore, a binding between one and more instances of virtual TPM and a single physical TPM that is able to allocate inside it registers from different virtual instances.

In conclusion, the proposed investigation and solution by this thesis enables the virtualization extension of Platform Configuration Registers protected by the TPM for a Deep Attestation use case, leveraging physical TPM NVRAM to store register values for different virtual TPM instances. The solution has been analyzed among other less suitable designs and chosen based on the environment in which it was proposed.

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem statement

Nowadays, **Cloud Computing** is considered a new enabling paradigm able to solve computing expensive hardware investments and operational costs. Cloud Computing changed the IT industry dramatically, reshaping the hardware provisioning and the infrastructure deployment. However, its potential made it become almost a requirement in the architectural choice, because of its efficiency and lower cost provisioning. If on the one hand, cloud computing has led to the cloud consumer the possibility of being able to take advantage of unique properties such as elasticity, resiliency, fast provisioning and multitenancy. On the other hand, security and privacy were negatively affected by the introduction of this new scenario, leading them to be a critical aspect in the overall architecture. Of course, many critical issues can be addressed with traditional approaches. In other scenarios, however, it is necessary to adopt cloud-specific approaches, precisely due to this paradigm shift.

Nevertheless, what both solutions have in common is that in most situations, they are software implementations. Therefore, to rely on a software solution it is essentially, first of all, be able to trust the system and the environment over which the solution is running, And it is in this situation that the Remote Attestation is involved. In theory, Remote Attestation is a method by which a **Prover** is able to authenticate the hardware and software configuration to a remote **Verifier**. Based on measurements taken during the life cycle of the Prover, the Verifier is able to determine the level of trust in the platform integrity. It is based on two major components: the Remote Attestation protocol and the integrity measurements architecture. The leading actor in the Trusted Computing Group with its technology called TC (Trusted Computing), based on the key enabler Trusted Platform Module.

Trusted Platform Module (TPM) is an international standard for a secure cryptoprocessor, tamper resistance, designated to secure hardware which implements primitive cryptographic functions that are starting point of more complex features that are built on them. Despite the potential of the TPM, it goes in stark contrast to what is the paradigm of cloud computing. Among the many functionalities, the

TPM has the task of saving within it aggregates of measures, which will then be used in the remote attestation process. The aggregate measurements are usually saved in special registers called PCR. Their primary use indeed provides a cryptographic record of the software state. The peculiarity of those registers is their update condition that is based on a one-way hash, which allows them not to be forged. They can be read to report the state of the machine and also signed for a more secure report. Here it comes the issue, a TPM implements only a strict number of PCRs, for example 24.

The virtual machines in the cloud environment are of a considerable number, and this would lead to having to tie each virtual machine to a physical instance of a TPM. This fact, however, would clearly disagree with the scalability that has been achieved with the cloud computing paradigm and would also prevent the possibility of migrating the VMs being cryptographically linked to the physical TPM. For this reason, several approaches have been brought to the literature to overcome this problem. From this problem arises the virtual TPM instances. However, they have at their base what was the main problem to which they tried to solve or that they are purely software solutions, going to lose all the potential of a hardware solution.

For this reason, the goal work of the TPM 2.0 virtualization extension project is to design a strong binding between different vTPM instances and a single pTPM so that cloud users can benefit from an hardware bound vTPM identity, which results more secure compared to a standard vTPM software instance. Moreover, migration can be now feasible, and in the Deep Attestation process, the chain of trust is finally extended to the hardware. Security and Privacy benefit from this implementation, in fact, vTPM is rooted in the physical Trusted Platform Module so that the physical platform itself protects its permanent state.

Within this project, the thesis work focuses on the investigation of PCR binding for **Deep Attestation** use case. In other words, how to save the various PCR belonging to different virtual TPM within a single physical TPM instance.

1.2 Objectives

The thesis work fits into the context of cloud computing and in the search for a hardware bound between a physical TPM and one or more instances of virtual TPM, focusing on the search for a binding of PCRs in the use of them within the Deep attestation. The basic idea of the project is based on the use of the memory space called NVRAM. It is the only memory space that seems to be possible to use to emulate the correct functioning of normal PCRs. This fact is due to specific design choices made by the TCG concerning the TPM implementation itself, which seem to be exploitable for the final intent.

The hardware bound sought within the project must emulate in all its functions the correct functioning of a real physical PCR, but above all, it must be capable of being used within the Deep Attestation process. The solution sought should in no way add excessive complexity to regular use of the TPM, the user of the solution, even if aware of interfacing with an instance of the virtual TPM, should be in a position to have the sensation of using a dedicated physical TPM. This fact includes bothe both the interface part, that is the responses to certain commands must be those expected, and from the point of view of security and privacy. In fact, they are the basis for the choice by a user of a cryptographic coprocessor such as the TPM.

The solution placing itself also within a cloud context must be scalable, and its use should not be a limitation, but, on the contrary, its adoption should bring about an enhancement. Once an optimal design has been found that meets the starting requirements, the solution obtained will be tested. In particular, it will be shown that the result obtained from a Quote operation, the critical point of the remote attestation and above all a part that concerns the use of PCRs purely, is comparable to a Quote operation carried out on physical PCRs not allocated in the NVRAM memory.

The search for an optimal design must, therefore, be accompanied by an in-depth study regarding the primitive functions of the TPM and how they can be used for the final project. The specifications of the TPM are indeed very restrictive for reasons due to its primary objective: security. Despite this, they are functions that we can consider elementary and usable as a whole to achieve the desired result.

The limitations to be kept in condition are not limited only to the primitives of the TPM and its architecture but also involve protocols defined on it and physical limitations of its implementation, such as a limited memory space.

1.3 Thesis structure

The thesis work is developed within two macro topics: **Cloud Computing** and **Trusted Computing**. For this reason, the implementation work must not be limited to a preparation study with respect only to the Trusted Platform Module that will be used but must also take into consideration the specifications of the original context in which it is located. For this reason, the state of the art part has been divided into two as the macro topics covered. The remaining part is focused on the development of the solution and how it is present in the remote attestation process. The thesis ends with a Proof of Concept concerning the design carried out concerning the PCR and how it fits within the Deep Attestation process.

In particular the Thesis is organized as follows:

- **Chapter 2:** In this chapter it is introduced the first macro topic, the *Trusted Platform Module*. In particular it will discussed the overview concept of Trusted Computing, the internal structure and architecture of the TPM and how they are involved in the specific use case of the Remote Attestation. The chapter, in fact, finishes with a brief introduction of the Remote Attestation concept.

- **Chapter 3:** In this chapter it is introduced the second macro topic inside which the solution is placed, the *Cloud Computing*. In particular it is explained the basic concepts of Cloud Computing and Virtualizaion. After this brief introduction the focus is concentrated in the cloud security, its challenges and how the trust execution environments are part of this new paradigm. At the end of the chapter the existing solutions and frameworks are addressed, analyzing point of strengths and weaknesses. In this way it is possible to have a more clear view of how the project is inserted in this environment.

- **Chapter 4:** In this chapter it is explained how the two macro topics are combined and how the PCR are involved in the Deep Attestation process. All the involved agents and their interactions are explained. In particular it is explained the design goal of the Deep Attestation process and its flow and how the PCRs and the hardware bound are involved in the process

- **Chapter 5:** In this chapter it is explained in more details the solution implementation. With a focus at a low level view. In particular it is explained the modifications that are made at the physical TPM interface, how the PCR life cycle is addressed and how the processes explained in the previous chapter are involved. The chapter finishes with the Proof of Concept of the project. During the PoC it is shown the hardware bound achieved regarding the PCR and how this is used to achieve the final Quote operation signed by the TPM.

- **Chapter 6:** In the last chapter it is summarized the obtained results and the possible future work on the project that could improve the overall design.

Chapter 2

Trusted Computing and Remote Attestation

Cloud Computing can be considered one of the significant trends in the last few years. Because of its fast development and reduced cost, more and more companies are adopting this architecture. However, this new solution has a large amount of security issues that have, for some scenarios, limited its adoption. All the information that we are exchanging over the internet can be considered as an essential digital asset and have an extreme value. For this reason, we are facing an increase in the spread of malware and enhanced hacker threats.

Therefore, in order to obtain and maintain confidentiality, integrity and authenticity of the information that we continuously share through our devices, multiple security techniques are adopted by enterprises. Against this scenario, the Trusted Computing Technology aims to ensure the security of the nodes that compose the network so that we would be able to establish a trust information transfer between nodes that have not been tampered. It is an upgrade in the computer architecture security and in order to achieve this goal, TCG has proposed a widely accepted solution by embedding the Trusted Platform Module (TPM) into a dedicated security chip [1].

In this chapter, the concepts that are the foundation of this new architecture will be introduced, in order to understand better the processes that are behind it and how they can be fully exploited to increase the security in a cloud environment.

2.1 Trusted Computing overview

The Trusted Computing had its roots in the middle of 1990s when the first solution of a systematical architecture based on a series of mechanism such as Root of Trust, secure storage and chain of trust was converging into a chip computer hardware. The main idea of the Trusted Computing is to establish a trust in a single platform to be sure that the software and in general, the node itself was not altered in any way. Then the trust can be extended between platforms and through the server with the usage of remote attestation, extending the trust finally up to the network.

As a core of TCG technologies, there is Trusted Platform Module (TPM), essential building blocks in a Trusted Systems. As in the introduction was said, there are plenty of software security solutions. However, all the security solutions that can be found in the literature do not take into consideration the not compliant status of the Operating System itself. Moreover, all the software solutions are subject to defects that can lead to potential breaches that can be exploited by an attacker.

Looking at this scenario, TCG aims to create an architecture in which it is possible to establish a chain of trust and through a proper attestation process, verify the full integrity of the system. All the possible applications and technologies developed with this aim are based on the TPM. Some of the possible solutions are the following:

- Enhancement in the digital signature process
- Identity management
- Secure execution of the applications
- Secure bootstrap
- Remote attestation

As it happens in the real-life also in this word, trust is a relative term, which means that we can trust some systems more than others. Accordingly to Trusted Computing Group[2]:

"A trusted component is one which is predictable."

So it is definitely not a synonym of excellent and secure, but it is undoubtedly a pillar over which we can build on. We have also to add that there are two main reasons to build trust and they are: a reliable evidence for the Attestation process and an out of band assumptions that we can make with the Root of Trust. Those concepts will be expanded later. Trust is usually not transitive, and in order to extend this concept further than the trust of a single terminal, we have to build a chain of trust step by step.

The basic idea is to establish at a first level the trust into a single endpoint then exploiting the remote Attestation establish trust between platforms and finally extend it to the network. Now that we also introduced the concept of trust we can understand better the six principles on which a fully Trusted System, that is a system compliant to TCG specification, is built on[3].

- **Secure Input and Output**. All the information inserted in the system that shared through the bus must be ciphered.

- **Endorsement Key**. It is 2048-bit RSA public and private key pair created by the manufacturer randomly at the chip creation time. Those keys will never leave the TPM chip in our case, and it will be used for Attestation and encryption of the sensitive data.

- **Memory Curtain**. The exact implementation are vendor-specific, but the general idea is based on an extension of the traditional memory protection techniques that provides the full isolation of sensitive areas that can contain for example cryptographic keys. In this way, even the OS does not have access to it.

- **Sealed Storage**. It is a procedure for the protection of private information by binding the data with platform configuration, hardware and software. It means that once that the data is sealed it will be released only with a particular combination of hardware and software. Primary usage of this technique is usually the DRM enforcing.

- **Remote Attestation**: is a method or better a process in which a third party can request a proof of trust combined with an identification. In this way, all the changes can be detected.

- **Trusted Third Party**: it was the main obstacle during the development of the TCG technology, and it is used to maintain anonymity during the trusting flow. From the version 1.2 of the TPM a new process called DAA (Dynamic Anonymous Attestation) was introduced, that introduced a new method in the obtaining of a certified AIK (Attestation Identity Key), core module in a remote certification. The main point in the DAA credential is that they allow the verifier to determine whether they are valid or not, once received from the TTP but they do not contain any unique information that can potentially expose the TPM platform.

2.2 Trusted platform module

The Trusted Platform Module (TPM) is a hardware component similar to a cryptographic co-processor, and it is the core in the development of the proposed solutions by the Trusted Computing. Before anything else, the Trusted Platform Module must be secure itself, and this is one of the two main aspects of its design goal.

The security itself means that it should be a Root of Trust for Storage as well as Reporting. In the first case, it should implement secure and reliable protection, confidentiality and integrity, for crucial data such as keys. TPM has not the capacity neither to resit to external tampering nor response, but instead, it is based on a concept that can be summarized as tamper evidence. The second central aspect is that the TPM must have all the functionalities that can be related to a trusted computing platform and Remote Attestation. All these functionalities depends on the key management and data encryption.

- **Platform data protection**: this includes data protection, basic cryptographic functions and data sealing.
- **Integrity storage and reporting**: integrity measurements are really valuable, and for this reason, they are stored directly in the TPM itself. Moreover, those measurements are useless if the TPM would not be able to report it securely using, for example, a digital signature during an attestation process.
- **Identification**: the reporting of measurement is a key point, and since we are applying a digital signature to it, the TPM must be able to manage identity keys.

Naturally, as we previously said, because of scalability reason and cost reduction, the trusted environment is not relying only on this chip itself but is the primary building block in a secure architecture. This concept means that all the high computational procedure are implemented externally.

2.2.1 Structure

The Trusted Computing Group establishes the exactly TPM architecture and all its functionalities. There are two different version of the TPM that was standardized and adopted, version 1.2 and 2.0 released respectively in 2003 and 2014. Naturally, version 2.0 introduced many updates, but the most important can be considered the interoperability with more cryptographic algorithms. Since the thesis relies on the latest and greatest version, we are going to introduce in this section, the significant specifications of version 2.0.

Components

Before the introduction of the physical architecture overview, it is worth to understand at the beginning the logical functionalities that TCG specified.

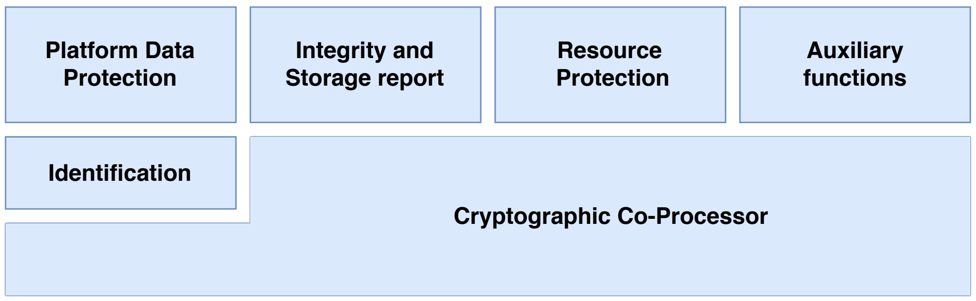


Figure 2.1. Logical TPM functionalities

It is worth to start with the two pillars in TPM functionalities. They can be considered the basic functions on which the more advanced functionalities rely on.

- **Cryptographic engine**: this system functionality includes functions such as encryption, digital signature, hash functions and random generator. The peculiar thing is that it has no external interfaces.
- **Identification**: it provides the management of certificate keys, an indispensable role in the remote attestation process.

The more advanced functionalities are built on the two previous basic functionalities.

- **Platform Data Protection**. It provides the integrity and confidentiality of all the critical data.
- **Integrity storage and report**. this is the basic in the construction of the chain of trust, a component at the centre of the remote attestation process. The TPM stores inside itself the measurement taken for integrity. Those measures are essential for the construction of the chain of trust since they incorporate trust.
- **Resource protection**. It refers to the access control to internal resources that are available inside the TPM. Auxiliary functions: The purpose of those functions is to improve the manageability of the processor, including all the functions needed to support the central functionality.

Internal Architecture

Now that we introduced the logical functions, we can understand better the internal architecture proposed by the TCG group. The major components can be viewed in the Fig.2.2, and they include first of all the I/O component, essential for the communication.

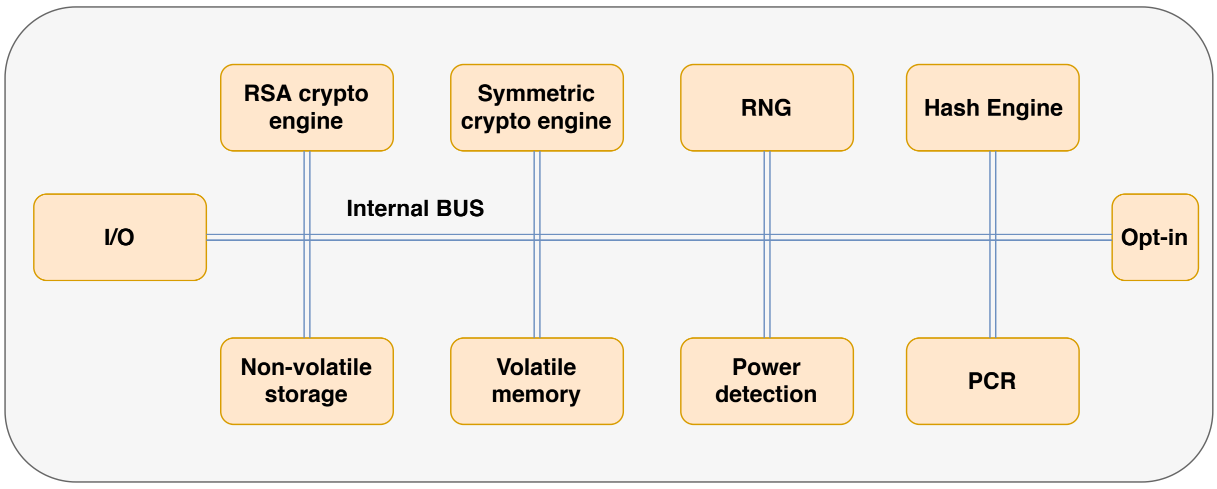


Figure 2.2. Logical TPM functionalities

Input/Output. It allows the management of the communication that comes from external entities to internal components and the opposite. In order to properly allow and enforce access control on the module itself, TPM is able to encode and decode the messages that have to travel through the chip bus.

Once that we are able to communicate with the TPM, we have access to many internal components, and everything developed to perform a specif action.

Opt-in. . This is the component that allows the checking related to the TPM routine, maintaining the storage and values of the related flags

RSA cryptographic engine. It is a component for the support of all cryptographic algorithms according to standard PKCS#1. The supported security levels are 2048, 1024 and 512 bits. All the algorithms are supported, such as encryption, decryption, digital signature and verification.

Symmetric encryption engine. AES supporting is the alternative key encryption algorithm. The mandatory mechanism is the one-time pad (OTP), also called Vernam-cipher, it is generated by the MGF1 key derivation algorithm.

Random number generator (RNG). Pseudo random generator.

Hash engine. Usually used with the PCRs as input and it is set accordingly to the standard FIPS-180-1, adopting the SHA-1 as the algorithm for the hash function.

Nonvolatile Storage. Dedicate memory inside the TPM, more flexible than the registries that are built-in, and that can be configured accordingly to the needs. However is primary function is the storage of the TPM persistent keys, integrity information and more critical application data.

Volatile memory. All the temporary data that is needed during the computation is instead stored inside this space of memory.

Power detection. Primarily used for the power state management, but it also supports physical presence assertions, critical for a physical signal application like DRTM.

Platform Configuration Register (PCR). Core part of the TPM for the scope of the thesis, those are a series of registers that memorize and store measurements. The PCRs are registered set up at boot time, and that later can not be erased. Their update is called extend operation. The extend operation is based on the following operation that is not reversible. The PCRs values can be naturally read to report their state, both internally and externally, also through a procedure called quote, which is an attestation. PCRs usage can also be implemented as a restriction in the authorization policy.

It is worth to expand the concepts regarding the PCR functionalities. At reboot time the PCRs are usually initialized to all zeros or all ones accordingly to specifications. Later on, every time that the TPM will update the PCR values, it will apply the following procedure:

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

As can be understood by this implementation, the extension operation is a single way operation, so it is not possible to extend the values to a derided one or to go back to a previous value. The number of PCRs that are contained inside a TPM is vendor-specific. Usually, PC client has 23/24 PCRs, and the usual allocation is the following:

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

As can be seen, every PCR usually has a specific allocation, and their number is fixed, this is why in the specification we find that is also possible to create inside the Nonvolatile memory (NVRAM) user-defined extend indexes that emulates the PCRs behaviour. The use case of PCRs are multiple. The two primary are Authorization and Attestation. The latter one is the one more interesting for our case study. It is a more advanced use case for PCRs since can not be achieved simply by applying a digital signature over a digest, but the agent requesting the proof will have to validate that the digest reported are matching the reported ones.

A TPM attestation is basically a proof of the software state that can be sent and offered remotely exploiting a cryptographic proof. This process will be fully explained in details in the next section.

2.2.2 Integrity Measurement Architecture

In order to achieve the final aim to have an endpoint defined as trust, we have to go through a process that can be divided into three parts: **Integrity Measurements,**

Storage and **Reporting**.

Integrity Measurements have the aim to create a chain of trust that is crucial to define the real state of a machine. The chain of trust is based on a simple concept: *every component measures the next one.* In order to securely store all those measures, as it was previously introduced, the TPM dedicated an internal memory for this purpose: the PCRs. TCG also designed the updating procedure in order to protect this mechanism.

- PCR are located inside the TPM
- At boot time they are reset to their default value
- Updating is done following the extension procedure

At the base of the chain of trust, however, must be defined, the so-called Root of Trust, which is a component that must be trusted by default since its misbehaviour is not detectable. The only thing that is possible to check in a Root of Trust is its implementation. Usually generally speaking the Root of Trust schemes include an hardened hardware module in which we can find three different Root of Trust.

Root of trust for storage - RTS

The usage of an asymmetric key pair defines the secure storage inside a TPM. Those keys are called Storage Root Key, **SRK**, and they are bound to the TPM users, which means that they can change once a new user wants to take the ownership of the TPM becoming the new owner. This procedure is called *Take Ownership.* From an architectural point of view, the key pairs manage a small amount of volatile memory where all the keys involved in encryption and decryption operation are stored.

There are two types of keys that can be involved in those operations, and they are the storage keys and the binding keys. The first type is the one used to protect the other keys, as it was mentioned before, generating a hierarchy for the keys. The second type the binding keys instead are used to cypher data and the symmetric keys that are handled by the TPM.

Root of trust measurement - RTM

As defined by the TCG group, the RTM relies on a piece of code which is outside the TPM scope, so it not stored or encrypted by the TPM itself. The Core Root of Trust is the component that contains the RTM code, and it is considered immutable. CRTM can consist of CPU instructions stored directly in the motherboard inside a chip or in alternative inside the BIOS itself. For the latter solution, there are two ways to integrate the CRTM inside it. It is possible to integrate it directly as a boot block, but in this case, the rest of the BIOS should be later checked and therefore measured. The second possible solution instead is to make the whole BIOS part of the so-called Trusted Building Block (TBB) and to achieve this solution the position of the CRTM is not rigidly defined The main objective of the RTM is to initialize the Root of Trust taking the measurements and updating the PCRs, passing the information to the RTS.

The measurement process at this step is crucial, and there are two ways to achieve this goal.

Static Root of Trust Measurement (SRTM) As the name is suggesting the trust process is entirely based on the immutable piece of code that we introduced before, the CRTM. In this solution, the CRTM measures the integrity of the BIOS or more, in general, the next piece of code in the boot-sequence, as represented in the figure. This measures is later stored into a PCR, and the control is now transferred to the next in the chain of trust. This ensures that the measures can not be faked since from now on, it is possible to compare the stored values with the one that is expected.

This approach has two significant drawbacks. The first one is that each component in the chain of trust depends on many layers. Hence we are facing with a scalability issue if we consider a large number of components. Moreover, in the modern operating systems, the components that compose the chain are not always executed in the same order, which leads to a problematic integrity measurement maintenance. The second major drawback is the called Time-of-check-time-of-use, a vulnerability that can be exploited in the time that passes between the measurement and the code execution. Moreover, using the SRTM, we are only checking the integrity of the machine at load-time. This means that it is possible to know and only trust what is loaded but not what is executed.

Dynamic Root of Trust Measurement (DRTM) In the TCG 1.2 specification, we find a solution to this issue, and it is indicated as Dynamic Root of Trust for Measurement. The main advantage of this approach is that the length of the chain of trust is substantially reduced. The whole flow is represented in the figure. The whole process starts with the now noted SRTM process. Now the platform is in a state that is called pre-gap. The platform is soon prepared through the DRTM Configuration Environment (DCE). Aim of the DCE preamble is to execute the Dynamic Launch Event, during this period the PCRs from 17 to 22 are set to their initial reboot state. Once that the PCRs are prepared the DCE makes the needed measurements, extending the previous PCRs bringing the platform in a state that is known as Dynamically Launched Measured Environment.

Root of Trust for reporting - RTR

The other crucial part of the process for trust is the reporting. RTR, in this case, is responsible for this purpose. It has two main functions. The first one consists on the display of the integrity measurement taken before. The second one, instead, the platform must be able to provide the measurement based on its identity. The TPM will sign the measurements, that are stored inside the PCRs with its Attestation Identity Key (AIK).

2.2.3 TSS Software stack

In order to provide a more flexible and scalable interaction with the TPM, the TCG group also defined a software stack. The TSS is a software to support the interaction between applications and the security hardware during the invocation of the TPM interface. While the TPM has its core functions, used for example for the key generation, TSS creates a trusted environment in which is possible to use those keys, to collect the integrity measurement and to allow the communication with the applications.

As it is explained in the TSS specification, the primary goal of this software stack is the following:

- Provide an API set for the communication.
- Ability to manage multiple applications.
- Manage the limited resources of the TPM.
- Provide the right and expected internal commands to the chip in terms of byte stream and parameter order.

With these assumptions, the TCG built an architecture to support this goal, creating a standardized interface. Naturally, all the parts that compose the TSS are OS-dependent, but their interaction remains unchanged.

The overall architecture is shown in Fig.2.3

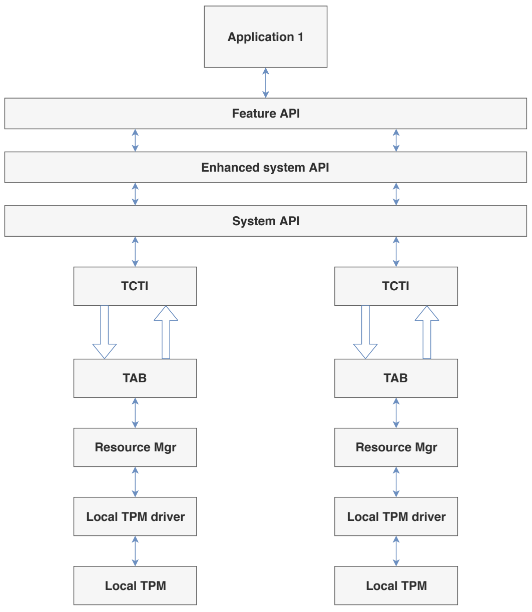


Figure 2.3. TSS

Each layer inside the stack has a different level of abstraction, the higher the level, the higher the abstraction. Not all the layers have the same capacity to interact correctly with the TPM, and for example, the Feature API covers more or less the 80 % of the possible calling functions. The deeper we go into the level, the more knowledge about TPM we must grant.

Feature API

As it was previously introduced the FAPI are the easiest way to interact with the TPM. The number of parameters that must be provided and the number of calls that must be done is significantly reduced. This goal is achieved using a user-selected configuration that creates a sort of default selection for what regards the cyphers to be used or the signing schemes, for example. If the profile is not chosen there is a pre-defined default profile.

Enhanced API

This is an intermediate level between the FAPI and SAPI, with the goal to reduce the complexity required for a single call, without however leaving the possibility to use cryptographic operations.

System API

One step below the ESAPI we find the System API or SAPI. Good knowledge of the TPM structures and functionalities is required. Both synchronous and asynchronous calls. It is developed with the aim to use all the low-level calls to the TPM

TCTI

This is a fundamental part in the software stack since until now it was only explained how to implement, at different levels, calls to the TPM. The TCTI is indeed in charge of transmitting the byte streams to the TPM.

TPM Access Broker (TAB)

TAB is mainly responsible for two functions. The first one is to control and manage the access to the TPM that could be shared between multiple processes. The second feature of the TAB is a security function. It must check that the processes have access to only those objects or sessions that own.

Resource Manager

The TPM has a limited amount of memory. For this reason, it has the need to swap from one session to another one, accordingly to the one that has been used at the moment. The goal of the RM is to provide the right resources at the right moment to the TPM. The peculiar characteristic of the RM and the TAB is that they are transparent to the higher layers. This assumption means that it is possible also to not implement them. However, in this case, it will be the application's task to set up the TPM for correct operation.

Device Driver

It is the last link that connects the application to the TPM. Its job is to send the buffer stream received to the platform and to forward the response back to the stack.

2.3 Remote Attestation overview

Remote Attestation is not a stand-alone process, but it relies on the basic functionalities as key management and integrity measurement. For this purpose, it needs the support of those essential functions that the TPM can quickly provide. In the previous chapter, we introduced the concept if integrity measurement and how the TPM is able to achieve this goal. What was not covered is the key management part. TPM can manage different keys; in total, they are seven. Each key has its own scope. For the aim of the project, the most critical key type is the platform identity key (PIK) that is the one used for the Remote Attestation. The PIK can be used only by the owner of the TPM. From this key, it is possible to create the Attestation Identity Key (AIK). This is key has a crucial role since it is the one that can be used to sign the internal PCRs value through the Quote operation.

Before proceeding with the explanation of the protocol adopted for the remote Attestation, it is useful to introduce the principle on which it is based.

Principle 1 - Fresh information To be focused just on the disk image is not useful if it is not taken into consideration also the running system.

Principle 2 - Comprehensive information The full internal machine state should be available to the measurement tools, and the attestation process should have the means to report target information fully. This principle can be considered as a vulnerability for the system since it could bring to disclosure of private information that could be exploited.

Principle 3 - Constrained disclosure Because of the disclosure of critical information, the attested machine must be able to choose which information should be sent to the verifier accordingly to a policy set. Moreover, the target machine should be able to identify the Verifier, attesting in its turn its current status.

Principle 4 - Semantic explicitness The whole attestation process is based on assertions, and for this reason, the entire semantic content should also have a logic form. In this way, the Verifier has the opportunity to make the correct conclusion relying on the received measurements.

Principle 5 - Trustworthy mechanism The attestation process and the architecture on which it is built should be trust by both parties, the Verifier and the target machine.

2.3.1 Protocol Model

The mean to achieve the Remote Attestation is the adopted protocol. There is not a single unified procedure to follow, but all the protocols have a reference model. The main characters are the Trusted Computing Platform that provides the necessary cryptographic functions, the remote Verifier and the Trusted Third Party.

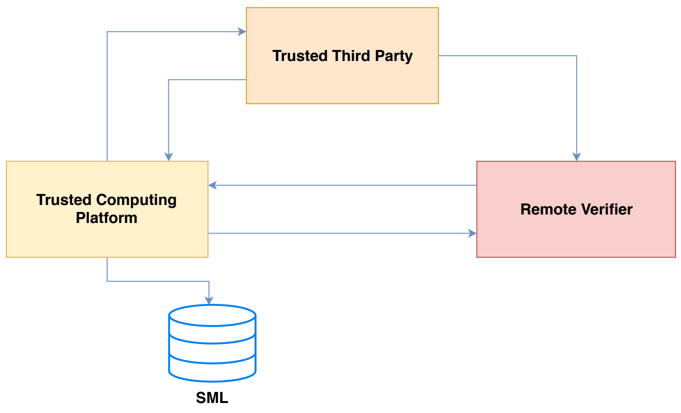


Figure 2.4. Basic Remote Attestation schema

The Host and the TPM compose the trusted computing platform. Together they can complete the Attestation of the platform integrity based on measurement and reporting.

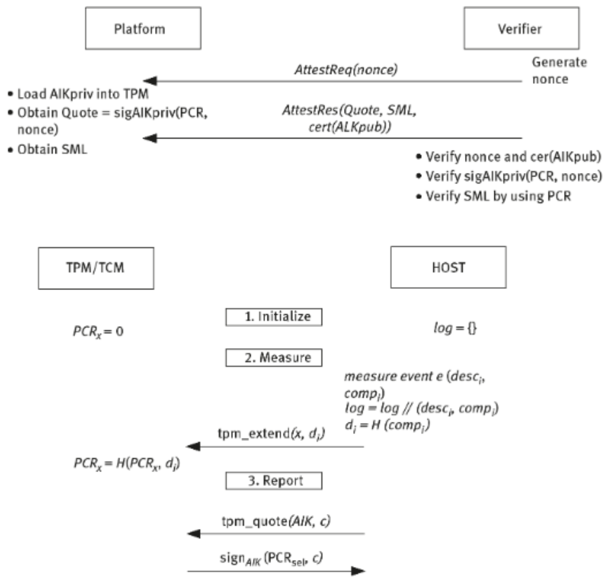

Platform
Verifier
AttestReq(nonce)
Generate nonce
• Load AIKpriv into TPM
• Obtain Quote = sigAIKpriv(PCR, nonce)
• Obtain SML
AttestRes(Quote, SML, cert(ALKpub))
• Verify nonce and cer(AIKpub)
• Verify sigAIKpriv(PCR, nonce)
• Verify SML by using PCR
TPM/TCM
HOST
1. Initialize
$PCR_x = 0$
$log = \{\}$
2. Measure
measure event e ($desc_i$, $comp_i$)
$log = log \,//\, (desc_i, comp_i)$
$d_i = H(comp_i)$
tpm_extend(x, d_i)
$PCR_x = H(PCR_x, d_i)$
3. Report
tpm_quote(AIK, c)
$sign_{AIK}(PCR_{sel}, c)$

Figure 2.5. Remote Attestation workflow

During the integrity measurement, all the measurements are collected inside the so-called Stored Measurement Log (SML) that do not reside inside the TPM, but they are still part of the TSS. The measures that the Verifier has requested will, therefore, be inserted into the TPM as a digest inside the PCRs. Once that the digest of the measurements is securely stored, it is possible to create a report that will be transferred to the Verifier. The file that is created is called Integrity Report, and it is a combination of integrity assertions and values. The first one identifies the measured component and the second one are the corresponding digests.

As previously mentioned, this is only the internal process to create the measurement report that the Verifier had previously requested. Once that the report is received the Verifier will proceed with the evaluation of it. The verification is based on the policies that are previously defined. The whole process is supervised by the Trusted third party whose role is to issue the certificate for the trusted computing platform and to provide verification of the authenticity of the TPM identity. In this scenario, the Trusted Third Party has also another role which is to provide to the Verifier the credentials that are needed to identify the Trusted Computing Platform.

Chapter 3

Security of Cloud Computing

Cloud Computing is one of the major trends in ICT in the last years. However, the concerns of insecurity and privacy violations are the limitation in its massive adoption. For this reason, it is essential to develop a fine trust model developed for cloud-specific scenarios to support the new trust metrics, and that can take into account the different users behavior in this environment. Despite this change of environment, security challenges not in the cloud are not so different except for the fact that the number of vulnerabilities has increased. Since the cloud environment includes several layers of abstraction [4], an attacker could compromise the service to each of these levels.

3.1 Basic of cloud computing and virtualization

The Cloud Computing paradigm changed the IT industry mutating the hardware provisioning and how services and infrastructures are developed. The main idea under cloud computing is to outsource the provisioning and management of hardware and software resources to third-party companies for a better quality of service and a lower cost in the delivery of resources. The National Institute of Standards and Technologies (NIST) defines the cloud computing as

> *"a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction[5]."*

The definition is broadly accepted and provides a clear understanding of how enterprises can use those resources to develop, host and control services on-demand flexibly, anytime that a resource is needed.

3.1.1 Cloud computing characteristics

The NIST inside his publication made efforts also to identify a unified view of Cloud Computing definition through its functionalities. Although the cloud environment is varied and very heterogeneous, making it very flexible in its applications. Five essential features have been identified to represent a Cloud Computing platform.

- **On-demand self-service**. Resources are not a permanents part of the infrastructure, and for this reason, they can be automatically provisioned when needed.
- **Broad network access**. All the resources in the cloud computing environment can be accessed and provisioned over the network through standards mechanisms. This characteristic allows the access to the services to heterogeneous endpoints and platforms as well as other cloud services.
- **Rapid elasticity**. Resources allocation is not fixed, and once they are needed, they can be extended. Moreover, two different extension operations are defined by NIST: vertical and horizontal. The first one means that more resources or a different computing capacity are needed. Horizontal resources, instead, concern the addition of services.
- **Resource pooling**. In order to reduce costs, resources are pooled for a better and more efficient use, which means that the physical layer is often shared between multiple users and services.
- **Measured Services.** The cloud environment automatically optimizes and control the resources. Moreover, those resources are monitored and reported for transparency usage between the provider and the consumer.

The previously presented characteristics are essentials for Cloud Computing functionalities. However, there are also two important aspects of the cloud computing platform that must be taken into consideration, and they are the *multitenancy* and *virtualization* characteristic. In a cloud environment, the resources are presented and provisioned to the consumers as virtual resources. Virtualization, in fact, enables the creation of those resources. In this way, multiple tenants are served from the same physical layer infrastructure. Here the second characteristic, *multitenancy.* It refers to the ability to serve multiple consumers from the same infrastructure.

3.1.2 Cloud computing models

Several Cloud Services have been proposed, but in particular, NIST defined three primary services:

- **Software as s Service**: SaaS.
- **Platform as a Service**: PaaS.

- **Infrastructure as a Service**: Iaas.

In addition to those primary three services, that will be covered below. It is possible to find a useful list of new cloud services proposition inside the ITU-T Y.3500[6]. They include a variety of propositions like Communication as a Service(CaaS), Data Storage as a Service (DSaaS) or Network as a Service (NaaS). However, all those categories are the result of a combination of the three main Cloud Capabilities types **SaaS**, **PaaS** and **IaaS**.

The three main models are universally accepted as necessary capabilities for cloud platforms, and they alter the separation of responsibilities in cloud operations between the consumer and the cloud service provider, as it is depicted in figure.

Infrastructure as a Service (IaaS)

IaaS is the solution that provides more freedom inside a cloud computing environment. The consumer has, in fact, the possibility to deploy and run arbitrary operating systems and softwares. IaaS provides raw access to an abstracted hardware that allows the consumer to build its own system

Platform as a Service (PaaS)

This type of service allows the consumer to outsource a great part of the computing and administration. **PaaS** provides an environment in which customer can deploy their own applications with the help of pre-configured tools such as runtime environments or programming languages tools. In this model, in fact, a complete virtualized environment is provisioned to the consumer, including operating system, web servers and databases. Using this service, however, decrease the level of governance that the consumer has over the system since now it is the Cloud Service Provider that must administrate the systems. Security at hardware level and OS level are now not over the control of the consumer.

Software as a Service (SaaS)

In this scenario, the level of governance over the system by the consumer is very low. As the name implies, it provides only software functionalities accessible in the cloud. However, it avoids the complexity of installation and maintenance of services that exists yet, and they should only support the consumer in his business applications. Example of those services are Microsoft 365, Google Gmail or Cisco WebEx.

3.2 Cloud Computing security

Cloud Computing has been promulgated as requisite in the wide choice of architectures based on virtualization and service-oriented. It was used as enabler for services that now can be provisioned at a lower cost and more efficiently. The main reason for its widespread adoptions is also due to some of its unique properties, such as elasticity, resiliency, fast provisioning and multitenancy. Despite these advantageous properties, Cloud Computing is a double-edged sword. Systems that are now offered at reduced cost became at the same time target and sources of malicious attacks. For cloud consumers, it is essential to take advantages of the Cloud Computing scalability, but at the same time, it is crucial to building an architecture that allows reaching the right level of trust over the service. Security and privacy are two factors that have an impact on all layers inside a cloud computing architecture. Therefore the security management is a critical aspect of the overall architecture. Different security treats differ for deployment model and for cloud service implementation. However, a significant quantity of those threats can be mitigated applying traditional security countermeasures. On the other hand, some of them require a cloud-specific implementation because of the newly adopted paradigm.

Cloud Computing security must be composed of a series a complex set of procedure, processes and standards adopted to mitigate information security in a cloud ecosystem. However, the massive concentration of hardware and calculation power inside a single ecosystem allows the cloud computing provider to have the potential of adopting robust and cost-effective defenses. Also, consumers can take advantages of this new paradigm because they can benefit of security resources that individually they could not afford. On the other hand, the concentration of resources and enterprises information inside a single location attracts attackers and gives them the motivation to spend time and effort to find vulnerabilities in the cloud infrastructure. Theoretically, the control of all the information can be gained through a single attack to the hypervisor. This process is called *hyperjacking*, for example.

3.2.1 Security challenges in the virtual environment

The new paradigm of Cloud Computing had introduced a division of roles for what concerns the operational security responsibilities over the different layers inside the architecture. On one side, there is the consumer that is responsible for the definition of security and privacy controls required. It is the owner of data and must protect the information and the system from unauthorized access. On the other side, there is the Provider that has the responsibility of implementing and providing security defenses in those layers to which the consumer has no access. The consumer is, therefore, conferencing an high level of trust onto the cloud infrastructure and more precisely onto the Provider. The primary security risks of cloud computing are presented by the Cloud Security Alliance (CSA) [7][8] and NIST[9] and are the following:

Data Breaches

A data breach is when sensitive and confidential information are released or stolen. Primary causes are the results of human error, application vulnerabilities or inadequate security practices. The security responsibility, in this case, is of both Consumer and Cloud Service provider.

Spofing Identity	✓
Tampering with Data	×
Repudiation	×
Information Disclosure	✓
Denial of Service	×
Elevation of Privilege	×

Misconfiguration and Inadequate Change Control

It occurs, then the misconfiguration of assets enables vulnerabilities. It can affects all the cloud service models and is purely the responsibility of the consumer. For example, In 2018, Level One Robotics exposed highly sensitive proprietary information belonging to manufacturing companies, including Volkswagen, Chrysler, Ford, Toyota, General Motors, Tesla and ThyssenKrupp. In this case, the misconfigured asset was a rsync (backup) server that allowed unauthenticated connections to any client.

Spofing Identity	×
Tampering with Data	✓
Repudiation	✓
Information Disclosure	✓
Denial of Service	✓
Elevation of Privilege	×

Lack of Cloud Security Architecture and Strategy

The moment of transition between a private architecture, in which the consumer has full control of the processes, to a cloud implementation is the one of the riskiest procedure. A lack of understanding of security responsibilities can lead to vulnerabilities. One useful takeaway in this procedure is to use cloud-native security solution to minimise the cost and risk.

Spofing Identity	×
Tampering with Data	✓
Repudiation	✓
Information Disclosure	✓
Denial of Service	✓
Elevation of Privilege	×

Insufficient Identity, Credential, Access and Key Management.

Cloud Computing fully revolutionized the standard Identity and access management, IAD and IDM. Therefore traditional **IDM** is not anymore sufficient, and a new paradigm has to be also introduced in this field[10] to take care of various aspects and challenges that also in the traditional identity management are a hot challenge[11][12]. As pointed out in the Cloud Computing Identity Management, cloud **IDM** has to manage dynamic environments with services that are on-demand provisioned. It is crucial in this phase to track and revoke accesses once it is necessary. Moreover, also, the key management issues are now critical requirements in order to deal with high-security risks[13]. It is needed, in fact, an efficient and robust key management. Also, the cryptographic algorithms and mechanism must be adapted to a cloud environment. Algorithms like MD5, AES or DES are consistently prone to security threats, for example.

Spofing Identity	×
Tampering with Data	✓
Repudiation	✓
Information Disclosure	✓
Denial of Service	✓
Elevation of Privilege	×

Insider Threats

In this case, the threats are outside the trusted customer domain. The Computer Emergency Response Team (CERT) is defining the insider attacks *"as the potential for an individual who has or had authorised access to an organisation's assets to use their access, either maliciously or unintentionally, to act in a the way that could negatively affect the organisation"*. The potential of this kind of threats is that insider attacker is operating under the trusted domain. For this reason it is essential to manage granularly the granted services. There is multiple solutions for that, using VLAN, access-list or security group tag. However, it is also vital to educate employee to prevent phishing attacks or to handle properly personal devices that can lead to breaches inside the system.

Spofing Identity	×
Tampering with Data	✓
Repudiation	✓
Information Disclosure	✓
Denial of Service	✓
Elevation of Privilege	×

Insecure Interfaces and APIs

Usually, in a cloud computing environment, services are exposed to consumer through a set of user interfaces and APIs. This implementation is mainly adopted to create a scalable interface to facilitate every consumer to deploy its own infrastructure. Therefore, if from one point of view, they are usually straightforward to use, they can be an open book to the internal design implementation. It is crucial the design of **UI** and **API** to granularly choose what should be exposed and what instead is not necessary to the user. All of this can be done through during the development phase for what regards encryption, encapsulation and abstraction.

Spofing Identity	×
Tampering with Data	✓
Repudiation	✓
Information Disclosure	✓
Denial of Service	✓
Elevation of Privilege	×

Weak control plane

The security and integrity of the data plane are the a crucial point in the system design. A robust control plane can prevent the system from leakage or data corruption, which can lead to a blind spot in the infrastructure design. The control plane in this scenario must support the complementary data plane, providing the expected stability. It is significant indeed an adequate policy enforcement, segmentation and a proper intrusion detection system, for example.

Spofing Identity	×
Tampering with Data	✓
Repudiation	✓
Information Disclosure	✓
Denial of Service	✓
Elevation of Privilege	×

Abuse and Nefarious Use of Cloud Services

This is one of the most significant challenges because it lead to our main problem: the trust of the infrastructure in which the service is running. Malicious threats can be hosted in cloud services. This attack design enables an extensive malware spread through a legitimate cloud service, used as a vector. Example of this is the recent Locky ransomware with all its variant which was using Drive services as vectors to infect endpoints.

Spofing Identity	×
Tampering with Data	✓
Repudiation	✓
Information Disclosure	✓
Denial of Service	✓
Elevation of Privilege	×

As can be seen, Cloud Computing is a very scalable solution that facilitates system design and architecture. However, at the same time, in this scenario, many security challenges are appearing. Cloud Service provider and the customer must work together, understanding their roles inside the architecture, and apply the right procedure and processes to prevent vulnerabilities and breaches in the system.

3.3 Trust Execution Environments in the Cloud Computing

Solutions, processes and procedure have been developed as a countermeasure to all these threats. However, there is something that is common to all those proposed solutions, and it is the fact that they rely mostly on software solutions. Therefore in order to be able to rely on software solutions, we must, first of all, be able to ensure that in the first place the software is correctly executed and that it gives the expected results as a result. Secondly, it must be taken into consideration that the system above which the software is loaded and executed has not been altered in any way. This fact would modify the result of security implementations, which means that at the root of the security of cloud computing there are no complicated procedures and countermeasures to deal with different vulnerabilities but rater the capability to correctly evaluate the reliability of a system, in order to guarantee the correct application of the solutions subsequently. All this would increase the level of security obtained that would not rely only on the software and the procedure implemented but would be based on the same reliability of the system. Of course this approach directly benefits all purely software implementations that now have a secure and controlled system in which to run. However, this does not bring improvements to the hardware attacks to whom a different approach is needed.

During the years, a lot of solutions have been proposed, all of them had in common the concept of Remote Attestation. The principle that has been previously introduced, in which a third party can define the hardware and software integrity of a platform. However, Remote Attestation is only a concept and not a standard, for this reason, it is possible to find different solutions with different approaches. Sometimes with procedures that do not require referer to **TCG** has a principal component. Since the proposed implementation has as enabler the **TPM**, it is worth to introduce the specifics of existing applications to understand the drawbacks and how those disadvantages can be overcome.

Before going to explain the different cloud attestation framework, it is useful to analyze the enablers of these designs. There are indeed alternatives to the TPM already introduced in the previous chapter and their aim is to create a TPM like trusted environment. They are **IntelSGX**, **AMD TZ** and naturally **TPM**.

IntelSGX

IntelSGX like TPM or ARM TrustZone is developed to match the requirements of the Trusted Computing Group. As the other enablers, InterSGX can create regions that are accessible only to high privileges levels. It was firstly developed in 2015 with the new Skylake architecture. Unfortunately, they are not widely used since they are supported only by a small amount of BIOS.

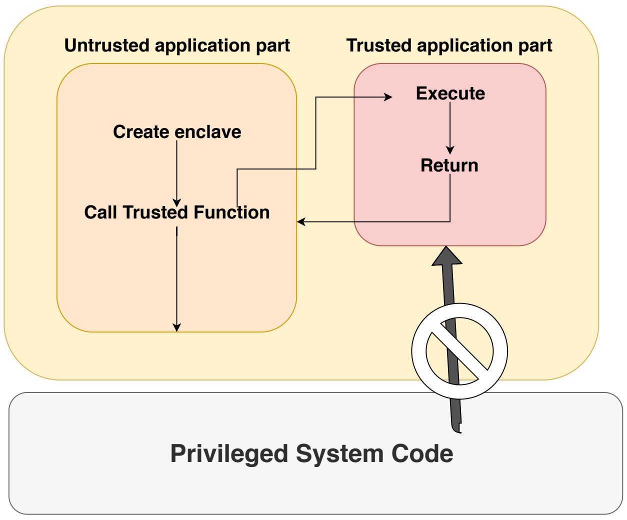


Figure 3.1. Application splitting

Its functionality is naturally similar to the TPM. Inside the memory, there are regions called enclave that cannot be accessed by applications. Every application is divided into two parts: one is the trust part, and it is contained inside the enclave, the other one, on the other hand, is untrusted and considered as unsecure. The enclave is the part that contains the private data and the code that runs over this private data. The SGX processor indeed protects the integrity and the confidentiality of the private data inside the enclave encrypting it. In this way, OS or hypervisors are excluded by this environment and can not access this particular region. Enclave region is fiscally separated in a part that is called **Processor Reserved Memory (PRM)**. Every application has assigned an **Enclave Page Cache (EPC)** which is a 4Kbyte to store code and data. Every enclave has its own EPC which is assigned by the OS. Now that it is known each component is possible to describe the workflow for an application:

- **Loading Stage**: CPU copy data inside the EPC pages and assign the pages to an enclave.
- **Initialized Stage**: CPU marks the enclave as initialized once all the needed pages are loaded inside the EPCs. Once the enclave is loaded is also cryptographically hashed by the CPU, in this way the system has a referment measurement. In this way, a remote party can understand if the enclave that is running the code has not tampered.

As can be noticed SGX enclaves leverages mostly in this strong encryption mechanism that is the combination of three factors: SGX Security Version Number (patch level) , device ID (unique 128 bit number assigned to the processor) and Owner Epoch (user entropy in the key generation). The combination of those three factors allows the remote party to attest an enclave and see if it has been generated by an SGX. The remote attestation feature is fully described in the Intel White Paper[14]. The intent pursued by Intel is, therefore, to use a new set of instructions to ensure that programs can execute their own code, or at least a part of it, in a secure environment that can also be attested remotely. Although this approach is still a double-edged sword as it allows the same malware to be able to run its own code in environments that are not accessible to the operating system and antimalware.

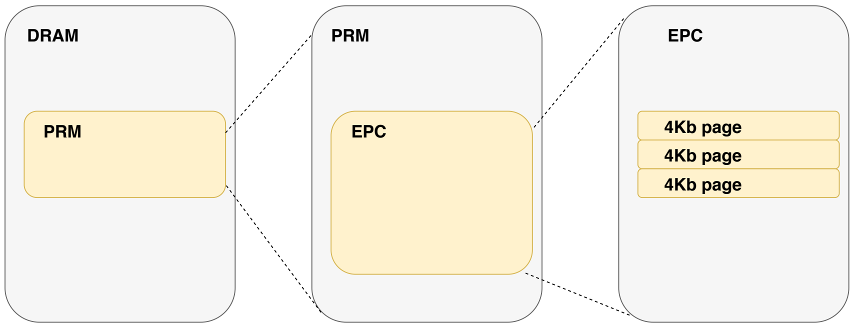


Figure 3.2. Enclave data structure

Moreover, if it is analyzed the SGX implementation can be seen that architecture is based on the Aegis Secure Processor, the first secure processor to not be vulnerable to physical reply attack, because of its Merkel tree construction[15]. To this architecture, SGX adds the possibility to change the Merkel tree generation dynamically. However, because of this design, it is sharing the vulnerabilities to access pattern leaks, which means that a malicious OS can perform cache timing attack against it.

ARM Trusted Zone

As we have already seen in the previous chapter and the last section, there are several Trusted execution environment platforms available for creating a context to rely upon to execute sensitive code. The last one is ARM TrustZone which is for its architecture design a better implementation not for cloud computing but for mobile devices and IoT.

ARM followed a totally different approach from TPM. While the latter one is designed with a set of predefined features set, TrustZone tries to overcome the little flexibility with a freely programmable trusted platform module. To do that, ARM introduces a security extension in the processors which provides hardware-based isolation between two context domains called normal word and secure word. The peculiarity of this distinction is that it is entirely orthogonal to the usual difference between kernel and user level. Moreover, the operating system is not wholly aware of it. The secure word indeed has higher privileges which leads it to access the normal world. However, the opposite is not possible. This concept is very similar to what was already addressed by IntelSGX even though it was then implemented in a different way. This role separation is under the check and control of the CPU, with the help of a memory partitioning. This partition is not concentrated only in the CPU, but it is propagated to the system. In such a way, once a software it is executed in the secure word as a completely different view of the system. The whole implementation is software-based, contrary to the hardware-based implementation of the TPM. Moreover, in TrustZone feature, there is a secure boot mechanism which ensures the integrity and authenticity of the system which is relying upon the secure word, that must be considered as a trusted environment.

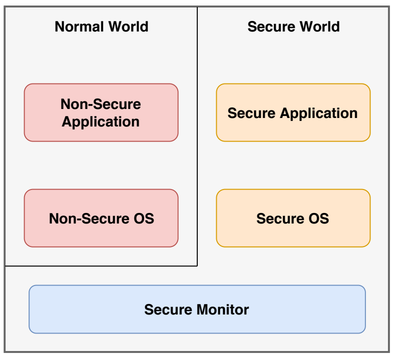


Figure 3.3. ARM TZ architecture

As previously stated, ARM TrustZone introduces the concept of a split word, in which CPU states can access the normal world, but the normal do not have privileges in the secure world. However, on top of this split world, there is a higher privileged mode called TrustZone monitor mode. It is responsible for the switching context, that is the software interrupts to change between the different contexts.

With this brief view of the general concept of ARM TrustZone, it can be obvious to think for this technology as implementation or even a virtualization replacement. ARM TrustZone can be viewed in fact as an alternative to the TPM functionalities or even as a virtualization solution. However, as per design implementation, the limit of the virtual machine would be only two. Moreover, the asynchronous execution of switching context is preventing a correct emulation of devices. Operating systems device drivers should be modified to support emulation, but unfortunately, this is not always possible. Proprietary versions of OS are not available if not in binaries. However, it can be considered as a potential candidate to replace the TPM. ARM TrustZone is much more versatile since a fixed configuration does not limit it, and it has potentially unlimited resources. Nevertheless, some applications need a secure storage mechanism which is not provided by TrustZone, and this lack is preventing the usage of it from being used as TPM directly mostly if applications need secure key storage. ARM TrustZone does not provide a canonical solution for remote attestation intrinsically, but can be however used to protect measurements for attestation reports.

All this, therefore, leads to a compromise: ARM TrustZone can, consequently, be used when dedicated hardware resources are not possible, and then the architecture of the solution must be based on a complete software implementation, which provides a mechanism to create a secure environment.

The Trusted Execution Environment is a context in which code and software can be executed at a higher level of trust because the untrusted system physically or logically separates it. As technologies enabler, there are multiple solutions. IntelSGX, ARM TrustZone and TPM are the more popular and primarily adopted solutions. Each of them has a different approach if designed to be used in a cloud environment in order to be able to support a remote attestation protocol in order to attest to a third party the trust of the system on which applications or services are running. For example, the first big difference is the approach relies upon the root of trust creation. Regarding the TPM, it resides directly in the ROM, and for this reason, the root of trust of measurements and authentication is the TPM itself. ARM TrustZone, on the other hand, does not provide a canonical mechanism, but the software implemented in its secure world can be applied in such a way to support its remote attestation protocol. In contrast, Intel SGX approach is focused on enclaves remote attestation provisioning, securely storing remote attestation keys in the processor. However, in a cloud, TPM provides the best functionalities. Even if its fixed set of features can be seen as a drawback compared to the flexibility that ARM proposes, it can provide significant security in the architectural implementation because of its purely hardware implementation. Moreover, it is not true that its fixed functionalities are limiting applications the TPM offers for the use of APIs that allow full access to the features of the same, without perturbing or having to implement applications differently as it happens when using ARM and IntelSGX. It is a crucial issue in a cloud environment which, by its nature, is a versatile and completely customizable environment. For a cloud environment, it is, therefore, necessary and better to use a security element totally separate from the system, more versatile from the implementation point of view even if it is only a tool and does not have a decision-making part in the trust decision.

3.3.1 Cloud attestation frameworks

As we could see from the alternatives to the TPM, the TC manages to guarantee security that is totally based on the hardware and is, therefore, a solution that can be considered stronger than the approaches that are based only on the hardware. Naturally, the solutions based on the latter are not risk-free. As also Trusted Computing presents critical issues and controversies[16]. Most important of all is the fact that the TCG does not have well-defined procedures for performing trust checks, but it is a set of specifications that offer a means to those who use them to implement their own solutions. However, this is also its strong point as it allows it to be versatile in an environment like the cloud. For this reason, it supports and

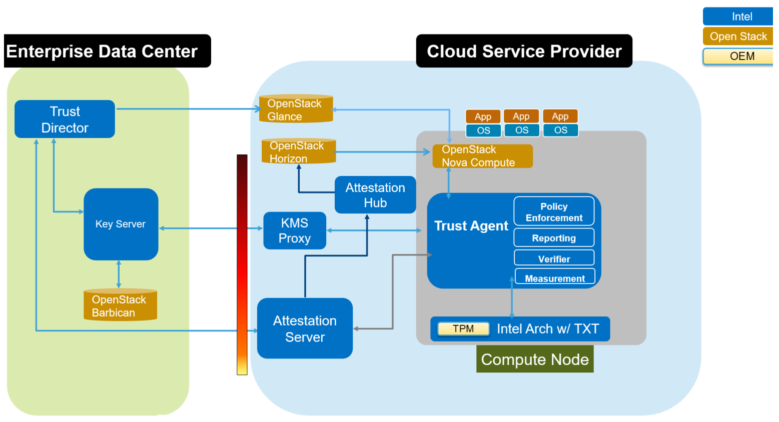


Figure 3.4. Intel OpenCIT architecture

is at the base of different attestation frameworks, which will be shown later here, in this section. The thesis project is aimed at improving the basic functionalities of the TPM in order to guarantee its better use in the cloud environment so that it is not, as in some cases, a limitation in the implementation, but a factor that can extend the functionality and potential of existing systems. The changes made, which will be explained later, are in fact aimed at being entirely transparent for already existing attestation frameworks. For this reason, in this section, we will illustrate the two main frameworks based on the TPM to understand their limitations and strengths compared in the cloud environment. Those two frameworks are **OpenCIT** and **Open Attestation Toolkit (OAT)**. Both frameworks relies upon the TPM and both of them can take advantage of the implementation that the work of thesis is based on.

Intel Open CIT

Intel Open Cloud Integrity Technology (OpenCIT)[17] is the Intel implementation for remote attestation of a cloud system, based on the architecture called Intel Trusted Execution Technology (Intel TXT). It allows to measure system components and attest that they have not been manipulated. The so-called Trust Agent, the unique application used for the remote attestation, is located inside the distributed machines.

The starting point for the remote attestation process is the Intel TXT. It is considered the Root of Trust for the remote attestation process. During machine boot, indeed, it will be the unchanging hardware measurements agent that will start the chain of trust measuring the first step of the boot process. Each software than involved in the boot process will measures the next one. Later through the now well know quote operation is able to pass through the TPM all the required information to the Attestation Server. The remote attestation begins with the server registration in the **Attestation Server**. The **AS** is the component in charge for the remote attestation process. It will compare the infrastructures measurements with the previous imported ones, also known as whitelist. At the registration, all the Know-good measurements from each machine are imported inside the **AS**. This process also includes the generation of the **Attestation Identity Key (AIK)**. The key pair generated by the host's TPM is for a secure attestation quotes transmission. The known-good measurements for each of the future attested machines are the basis for the later integrity checks. The results produced by the Attestation Server will be reported though the **Attestation Reporting Hub** to a service known as Open Stack.

The trust agent is another critical component of the architecture. It is part of every physical server, and its aim is to enable the remote attestation with the AS and to extend the chain of trust. The Trust Agent is responsible of the communication with the TPM for the secure quote operations, but it takes part also in the VM or Docker container images creation. It intercepts the requests indeed before they are sent to the hypervisor. Regarding the VM generation, they are under the control of the Trust Director, as well as the Trust Policies, docker images and encryption of VM images. The Trust Policies selection is a peculiarity of Open CIT which allows defining the folders and components to be explicitly measured and later attested, though precisely a Trust Policy. Those policies are digitally signed by both the Trust Director and the Attestation Service. Regarding the encryption of VM images instead, two other major components now are involved: Key Borker Server or also known as **Key Management Server (KMS)**. It handles the keys used to encrypt and decrypt VM images, which only happens if the Attestation Server involved in the remote attestation is considering as Trust the outcome of the attestation based on the previously established Trust Policies.

Host Trust Attestation is the process of verification based on known-good values previously imported. Those values are precious since, through them, the AS is capable of detecting deviations from the expected host behaviour. The policies, as earlier explained, are files and directories that are measured by the Trust Agent component during the boot time. Each measurement taken will be later saved inside the TPM registers. Once the server is completely booted the results of the booting process is sent and compared with the *known-good* measurement by the AS. The set of values called *known-good* is incorporated inside a structure: whitelist or **Measured Launch Environment (MLE)**. Each **MLE** contains first of all the platform-specific known-good, BIOS MLE. Also, it includes the so-called VMM MLE. Those are measurements that are not related to the host but concern the OS, kernel and other components like the hypervisor. Every PCR inside the TPM will have its own definition inside the flow. They are used to store the aggregate results of the machine measurements, and their values will be compared with the known-good conditions. Different PCRs have different measurements parameters; for example, PCR 0 is designated to BIOS ROM and Flash Image. Every PCR is indeed involved in one or another MLE accordingly to the setup.

However, as it can be deducted by the whitelist management, Open CIT has some limitations regarding the servers that are frequently updated. The update procedure introduces an overhead that is represented by the reimport of updated whitelists. If this does not happen, it will bring to a system integrity break. Moreover, it requires a compatible processor for the support of the Intel TXT technology, which represents the Core Root Of Trust in the attestation process.

Open Attestation Toolkit - OAT

Open Attestation Toolkit (OAT)[18] is another common framework developed by Intel in 2010 that can be used to implement Attestation Service in the infrastructure. It is able to measure and reports the host status of platforms that are supporting TPM as key technology enabler for the architecture. Intel OAT is fully compliant to the TCG specifications only up to version 1.7. Its key features are the supports of major linux OS, and it has a PCR-based report schema and policy rules. Moreover, OAT has a major number of RESTful based API to interact directly with the host agent and the Appraiser and with other tools that can be easily integrated. For example, the final user can use API queries to request attestation of a trusted platform or to interact through the whitelist API directly with the component.

A simple schema of the basic components of OAT can be viewed in figure 3.5. It accurately reflects the specifications set by the TCG. The host agents are running inside every trusted platform. Their role is similar to the OpenCIT trust agent. In this case, there is also a privacy CA which its aim is to provision and certificates each Host Agent AIK. The database responsible for the collection of the trusted known PCR values is called Whitelist Database, and it is naturally under the control of the Attestation Server that will use those values for the trust check validity over the reported measurements during the attestation process.

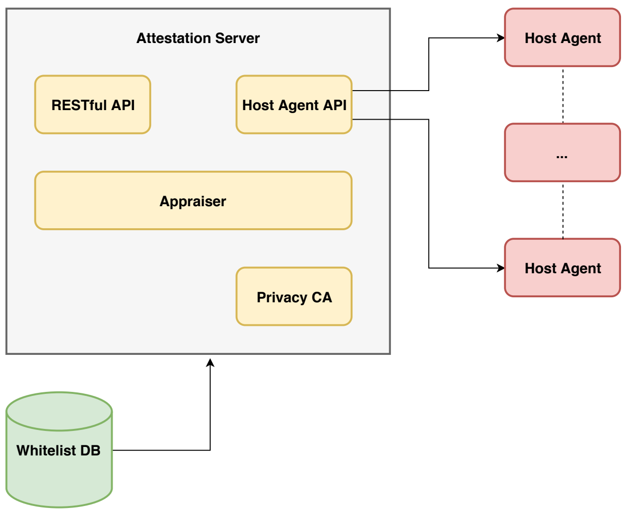


Figure 3.5. Open Attestation Toolkit architecture

When the Attestation server stars the attestation process, the Host Agent is in charge of creating the called Integrity Report. Two different sections compose the

IR. In the first one, all the PCRs values related to a trusted platform are reported. In the second section instead, all the measures performed by the Host Agent are included. Both sections are useful for the validation of the host status. The validity of each measure is done comparing the report with the white list database. Until version 1.6, the validation approach was slightly different, causing an inefficient interoperability with systems that are cable of supporting IMA. From version 1.7, it is, in fact, possible to skip the whole PCR validation and use only PCR number 10, that is responsible for maintaining the aggregate results of IMA measurements over the system. With the interoperability introduction of IMA is now also possible to attest not only statically a VM or a trusted platform but a run time attestation process can be guaranteed. Major drawback of OAT is that is supporting only TPM 1.2 despite the presence of a more recent release (TPM 2.0).

Keylime

Even if OAT and OpenCIT can be considered as two milestones in the field of Remote Attestation, recently a brand new framework is becoming more popular: **Keylime**, born in a research team in MIT's Lincon Laboratory. Like its predecessors, Keylime could be a TPM based remote boot attestation and runtime integrity solution. Its strong points are the following:

- **Full compatibility** with the Linux TPM2 Software Stack, on which it is built on
- **Open source project**. Like all the open-source projects community contributors are encouraged to make changes and improve its functionalities.
- **Accessibility**. Differently from other projects and implementations, the understanding of low-level TPM's operations is not necessary, and this is applicable to both developers and users.

The keyword in Keylime project is building trust in cloud computing. It is an end-to-end solution based on the period attestation of nodes, extending the TPM capabilities to the cloud environment, with always the same goal: allow the tenants to be able to check that the applications and the whole infrastructure over which they are running has not been tampered with.

The two technologies that are the foundations of this framework are the **Trusted Platform Module** hardware and the **Linux kernel** subsystem with the Integrity Measurement Architecture (IMA). The point of strength of Keylime is its scalability because of the performance of its architecture. The hardware TPM calls are reduced to improve performance, to do this it implements the *vTPM* quote operation. In this way, the number of Virtual Machines that can be managed increases considerably.

As it was previously introduced Keylime secure the cloud environment in two different phases, the Trusted Boot and the operational trustability of the infrastructure. Keylime is based mainly on three components: **Agent**, **Registrar** and the **Verifier**. All of them are implemented with the goal to achieve a high level of security but with particular attention to the performance as well. In addition to those components in the tool kit of Keylimme, it is possible to find also the Keylime **Tenant** which through a *RESTful interface* for the communication of all the components. However, another critical component is the included **Keylime Certificate Authority**. As it was already discussed in the thesis, it is a fundamental part of the trust established between the nodes and all the components involved in the remote attestation process. The *Certificate Authority* plays the critical role of provisioning all the keys on which the system relies on. It is also responsible for the revocation of certificates in case of security breaches in the trust relationship between the components. A more general overview of the role of the certificate authority is, however given in a dedicated section of the thesis, and the Keylime CA is built on the same concept.

However, to support the trusted workflows, the main characters and their roles are the following:

- **Keylime Agent**. The agent is installed in every node in the infrastructure. Its primary function is the interaction with the Trusted Platform Module. Those interactions take part in two different phases of the *Remote Attestation*. During the collection of the Integrity measurements, it is responsible for their communication to the other components to start building the chain of trust. In the same way, it is responsible also for the request of the quote over the collected measurements, once those are requested to verify the trustworthiness of the node.
- **Verifier**. Its role is the continuous check of the different nodes through the quotes provided by the Keylime Agent. Through the measurements received in the quote operation, it is able to determine any manipulation in the infrastructure system.
- **Registrar**. During the attestation, process keys are naturally involved in attesting the ownership of the data exchanged. Registrar is responsible for maintaining an updated known secure public keys list. Moreover, it has an important role in the process since it is also the phase for the storage of the hardware TPM key. As already mentioned in the first section of the thesis, the hardware TPM key verifies the validity of the TPM itself, crucial in the process because of its role of the root of trust. To keep an updated list of all the keys, the Agents, once deployed are asked to register itself along with their initial state

Now that all the components have been introduced is possible to understand better how effectively Keylime is supporting the Trusted Workflow. In addition, it is needed the introduction of also concepts that are related to the keys used during the process. The attestation Keylime workflow is based on the following keys:

- **Endorsement Key (EK)**. It is a hardware root of trust key which is burned by the manufacturer inside the TPM at the moment of the production. It can be considered as a unique identifier of the TPM. Because of its nature, the private key can not be changed or erased. The manufacturer instead publishes the public portion of the key.

- **Storage Root Key (SRK)**. Differently from the EK, this key is instead re-generated every time that the TPM is reset. Its primary function is to protect the AIKs.
- **Attestation Identity Key (AIK)**. The AIK is used instead to sign the attestation quotes generated by the TPM.

In order to support the Keylime workflow, some prerequisites are needed, especially regarding the BIOS and the operating system. For what regards the BIOS, it must support TPM's compatibility for the measurement of firmware and bootloaders. In addition, the OS should support the runtime measurements of the application, which can be translated in a compatibility with Linux IMA or Policy reduced IMA.

Once all the prerequisites have been respected, it is necessary to initialize the various components of Keylime. These components must be initialized in a precise order for the workflow to be carried out correctly. First of all, the components is the Registrar. It is deployed by the tenant, which also attest its integrity state as the root of trust. Naturally, the Registrar can be located in two different places: tenant infrastructure and Cloud infrastructure. Both scenarios are supported efficiently. At this point, once the Registrar has been fully initialized can start to accept Agent registration. This phase includes the TPM AIKs storage.

In parallel, another component is deployed, even before any other nodes. This component is the CA. It is responsible for the signing of keys sent to the nodes.

The final step is instead to deploy the nodes and provision the nodes in order to boot and verify them. Once the node is up and running and verified can be successfully registered with the Registrar.

An interesting fact regarding the Keylime is that it is supporting two different approaches regarding the attestation process. One can be considered more classic since it is supporting the standard Trusted Boot workflow. On the other hand, Keylime is also supporting a novel approach for the trusted workflow, called **Three Part Key Derivation** (**TPKD**), that can be summarized at a higher level as follow:

- The node is registering the public AIK with the *Registrar*.
- The *Tenant* generates a key and cryptographically split it in two halves. One of the two halves is given to the target node to start the provision with this particular node and to check with the Registrar that the AIK is also valid.. The other half is instead kept by the tenant or given to the CV in case of delegation for the integrity check of the node.

- In this way *Tenant* and *Verifier* can send separate attestation requests to the *Agent*, using the Registrar instead to validate the quote. The node, in fact, provides a quote to the CV which can check the integrity of the node and provide later its half of the provisioning secret to the node in order to complete the process.
- At this point, the node should have both halves of the secret if the integrity verification from the *CV* was successful. In this way, the node can obtain its provisioning key.
- Once the bootstrapping is completed, the *Verifier* can requests quotes to the *Agent*. If the quotes results to be not valid, the CA is notified for a renovation and invalidation of the keys.

3.3.2 Drawbacks of existing applications

In the previous section it was shown that there are multiple way to build an infrastructure for the attestation of trusted platforms. However, for VMs that are running on top of the hypervisor is difficult to provide them with the proper requirements for the remote attestation process. This is due to the fact that all the framework expect the usage of a dedicated TPM. Scientific literature has widely discussed how to extend the Root of Trust of Storage and Reporting, which is usually covered by the TPM.

One of the most popular solution is to deploy a virtual TPM instance simulated via software. In this way every virtual machine has its own private TPM. A shared hardware TPM would not be feasible since the number of PCR is limited. The hardware TPM in this new solution has now a different role and should be involved as Root Of Trust of Measurement. Some of the PCRs can be devolved to the management of vTPM. However, all the registers inside the vTPM have a major drawbacks which is that they are completely software developed. To fully exploit the potential of the TPM as a hardware resource it is necessary to have the same intrinsic security in the PCR of a TPM hardware but also to be able to use the fact of being able to use and have a private TPM for each VM. For this reason in the thesis project a way was found to be able to map each so-called virtual PCR within a hardware TPM. Thus, the binding created between a virtual TPM instance and the physical TPM would allow us to continue using TPM-based remote attestation systems compliant with the TCG and at the same time be able to tp extend the chain of trust up to the hardware level. It would also resolve the TPM reset attack which suffers software implementations purely. The solution is also introducing little overhead in the entire process since it is transparent to the Application Server.

Other existing solutions, on the other hand, proposed to integrate inside the hypervisor a component for the virtual machine remote attestation. The component is called Integrity Verification Proxy (IVP). In this way, the classical RA process can be used since the VMs integrity relies upon the system that is monitored. However, the solution is expecting the VM to run in debug mode creating an overhead and bottleneck in the VM performance[19].

Chapter 4

Deep Attestation process

The whole Deep Attestation flow is depicted in Fig.4.1. Every element and phase will be discussed in deep in the following chapter. More precisely in this section the Deep Attestation protocol will be described at an higher level, while in the next section it will be explained in more details the actual pTPM implementation and what changes are made to the interface to support the following protocol.

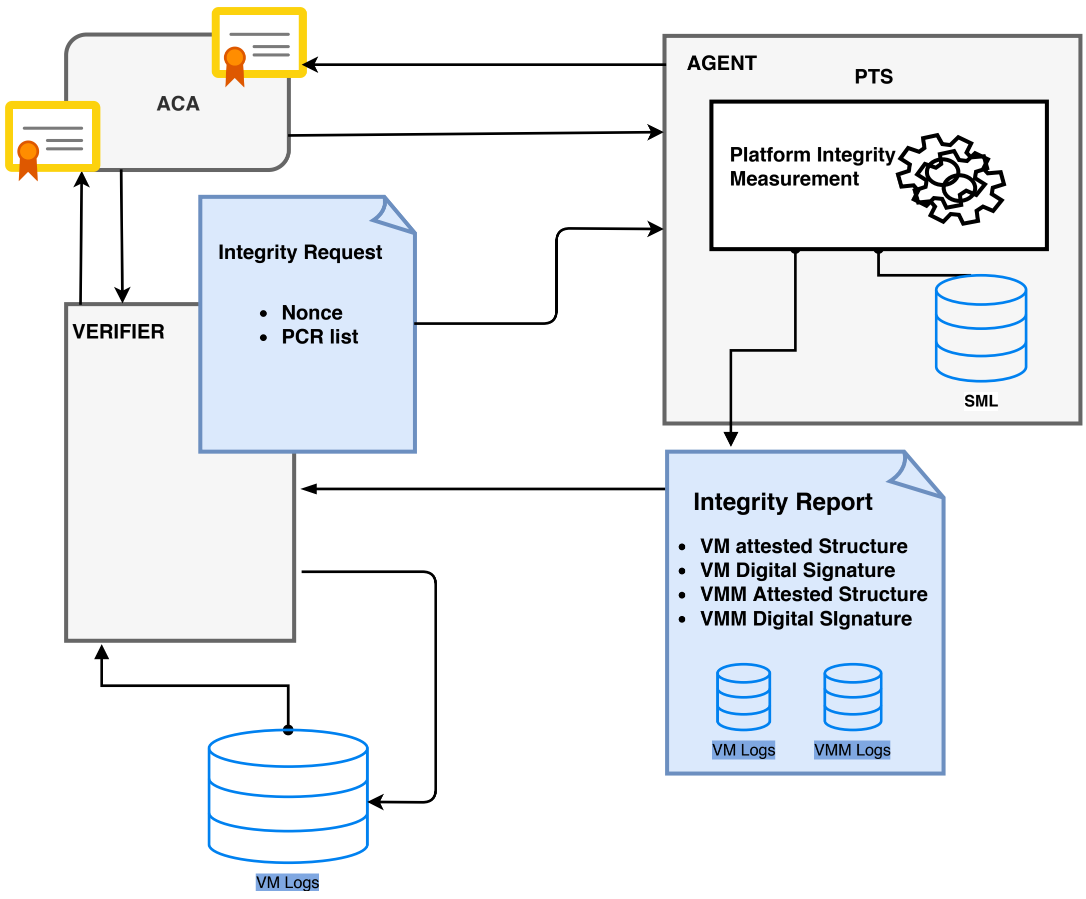


Figure 4.1. Deep Attestation flow

4.1 Architecture design goal

In this chapter, we are going to explain in more in-depth details the architecture for the realization of the TCG-based Deep Attestation. The proposed technique aims to verify the status of a Virtual Machine, that is the trustworthiness of the software and storage. Considered the infrastructure in Fig.4.2, the integrity of a VM relies on two components: **Virtual Machine (VM)** and **Virtual Machine Manager (VMM)**.

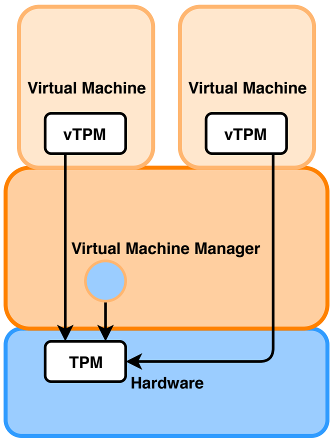


Figure 4.2. Logical Architecture dependencies

To provide reliable proof of integrity during the lifecycle of the virtual machine, the attestation of those two components should be made separately.

- **Attestation of VM**. This attestation can be subdivided into two phases, accordingly to the state of the machine. At booting time ensures that only expected programs and configuration files are loaded inside the VM. However, at run time, the attestation process extends the trusted boot, providing a controlled and attested execution environment.
- **Attestation of the VMM**. Virtual machines rely on this node, and for this reason, a proof of the integrity of the stack that is manipulating the machines must be provided, this also includes the aspect of a genuine virtual storage.

The last aspect, the virtual storage, depends on the cloud infrastructure, and it could be deployed as a separate node. In this scenario, the iterative attestation to reach the full Deep Attestation process is composed by three-layer. Additionally, the Secure Manager will be attested but with the same iterations of the VMM. For this reason for the thesis, it has been chosen to integrate the virtual storage as a part of the VM; and the storage controller as a part of the VMM.

As it is said, the Deep Attestation process is composed by two or three attestation layers. In order to reach a scalable attestation process, an iterative attestation schema has been adopted. It means that the Verifier initiate a single attestation session for the VM, but the Integrity Reporting incorporates the two different attestations, respectively the integrity of the VM and VMM.

Now that the aim of the protocol is defined, the various components in Fig. 4.1 and their purpose within the protocol will be introduced. More precisely the following components are part of the Deep Attestation process:

- **Verifier**. The Verifier is the remote party that triggers the start of the Deep Attestation process. Its purpose is to decide whether a particular system (Prover) can be considered *Trust* or *Untrust* based on the information received from the client.
- **Prover**. It is the system considered in an *Unknown* state at the beginning of the process. It responsible to collect *Attestation Data* during his life-cycle and provide them later to the Verifier.
- **ACA**. The *Certificate Authority* is the third party trusted entity responsible for issuing the certificates. The CA is trusted by both parties: Verifier and Prover.

Each entity described above will take part of the process in different phases. The whole process can be summarized in the following phases:

- **Resources allocation**
- **Identity credential enrollment**
- **Platform Integrity Measurement**s
- **Integrity Request**
- **Integrity Report**
- **Report Evaluation**

4.1.1 ACA - Attestation Certification Authority

Certificate Authority is the entity responsible for issuing digital certificates to the **Prover**. Through these certificates, a relying party, in our scenario, the **Verifier**, can rely upon the digital signature made by the Prover over the Attestation Data with its private key. Key that is bound to the issued certificates, which follow the format specified in the X.509 standard.

It is essential to remind the indispensable role of the CA during the attestation process for the particular reason that the issued certificates provide some degree of trust over the Prover. The ACA is involved because it has to prove that a specific Prover inside its implementation (in particular in its TPM) is presenting the private key associated to the certificate. The Certificate Authority moreover must provide several services that are directly related to the certificate life-cycle.

- **Certificate Request**. The CA is expected to have a robust validation process for what regards the private key used by the Prover during the digital signature.
- **Certificate Renewal**. There are scenarios in which certificates usually has very long life, others in which instead a short-life cycle is more appropriate. In the last scenarios the renewal process is essential.
- **Certificate Revocation**. There are a lot of reasons for which a certificate should be revoked and they are strictly connected to the renewal process.

4.1.2 Verifier

The **Verifier** is an appraiser. It aims to decide on the **Prover**. The decision-making process is the evaluation of the Prover against the *Verifier standards*, represented by its *whitelist*(database collection of the intersted measures over the Prover). The **Attestation Data** supports the trust decision made by this agent requested on behalf of the Provider. They are used by the Verifier to judge and complete the full decision-making process over the hardware and software integrity.

The **Integrity Request** supports the collection of information, that is made at the beginning of the Attestation Process. Later, based on the Attestation Data received in the **Integrity Report** by the Provider, the Verifier can compare its expected behaviour with the one reported. The latter process is called *Report Evaluation.* The functions of the Verifier can therefore be summarized as follows:

- **Integrity Request**. During this phase the Verifier sends a request to collect the necessary Attestation Data.
- **Report Evaluation**. During this process the Verifier compare the Attestation Data with the expected behaviour represented by *known fingerprints* or *whitelists.*
- **Signature Verification**. In order to trust the received Attestation Data, the Verifier must procure the certificate corresponding to the Prover to verify the integrity of the data received.

4.1.3 Prover

The Prover is the core entity of the entire Attestation process. Prover is responsible for different aspects of the attestation protocol, depending on the phases. The Prover is, in fact, the target of the attestation protocol. In order to provide the right information, the Prover must therefore first be provided with a certificate related to the key that it will use during the attestation. Applying a *digital signature* to the Attestation Data will provide different services to the Verifier: **Data Integrity, Non Repudiation and Sender Authentication**.

The Attestation Data, on the other hand, can be collected only if a proper **Platform Integrity Measurement** process is implemented. During this phase, the Prover must collect the measurements that will be sent inside the Integrity Report.

The functions of the Verifier can therefore be summarized as follows:

- **Identity Credential Enrollment**. During this phase the Prover has to prove to an ACA the possession of a key that will be used to sign the Attestation Data.
- **Platform Integrity Measurements**. The Prover has to collect the Attestation Data measurements.
- **Integrity Reporting**. Once that the Verifier sends its Integrity Request, the Prover must collect the Attestation Data and send all the information signed to the Verifier for the Report Evaluation.

4.2 Deep Attestation flow

Now that the general architecture structure has been introduced with all its entities. It is possible to describe more in details how those entities interacts to support the Deep Attestation process.

The remote attestation protocol alone is a valuable solution for attesting the integrity and establish unauthorized hardware/software modification in the Prover. However, in a cloud environment, the Prover is represented by different layers and entities. For this reason the VM that is the final target of the Verifier usually does not have privileged access to the Physical Platform (hardware layer). It is, therefore, impossible to verify the complete integrity of the VM if also the other entities inside the Prover are not attested. In this scenario, the Deep Attestation protocol extends the capability of a classical Remote Attestation process employing the virtualized allocation of TPM (vTPM) and their binding with the hardware TPM. The overall structure is a three-layer architecture:

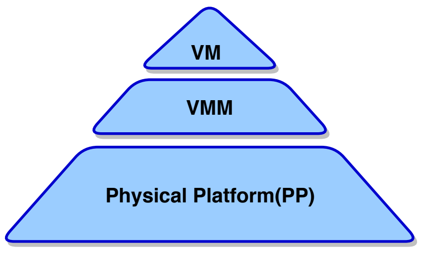


Figure 4.3. Three-layer structure

As can be seen from the Fig.4.3 the Prover, that from a point of view of the Verifier is a single entity, is however structured in three layers. At the bottom of the layered structure there is the Physical Platform, which includes one or more hardware TPM. It represents the RTM in the Deep Attestation protocol since all the measurements will be stored inside registers(PCR) allocated in a memory space accessible to only the pTPM. At a higher level, there is the Virtual Machine Manager. It includes a VMM attestation agent, which is responsible for the extension of the remote attestation process. The VMM is also responsible for the correct communication between the pTPM and the virtual instance of it. At the top layer instead, we have the VM instance.

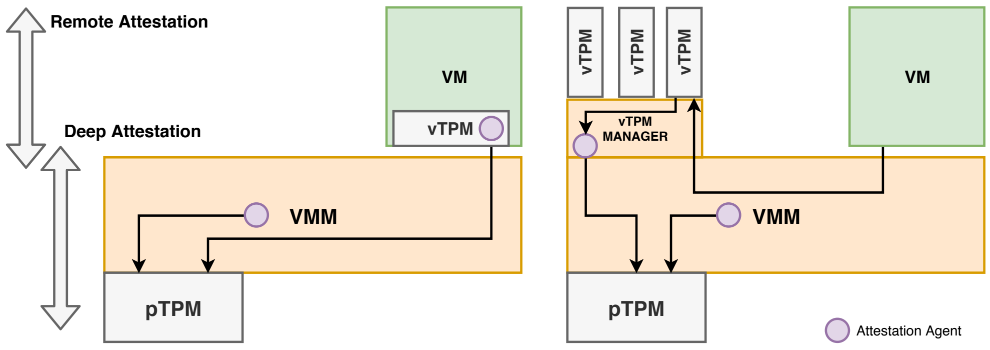


Figure 4.4. Logical attestation agent provision

Accordingly, to the various cloud infrastructure, the Prover structure can have different implementations. The vTPM entity can be an instance inside the VM, or a separate VM can manage it. The first case has a high provisioning cost since every single VM deployed will require a single attestation agent and the communication with the physical TPM is critical, because of the lack of privileges to access the Platform Layer. The second option instead has a reduced provisioning cost, running a single attestation agent, capable of communicating to the Physical Platform. The remote Verifier, indeed, will be able to attest through a standard remote attestation protocol the integrity of the upper layer that has a strong binding with the physical hardware thanks to VM manager instance. However, during the Integrity Reporting phase, not only the integrity of the VM will be attested, but instead, a report on the VMM will be integrated, so that to extend and complete the DA.

vTPM PCR binding to pTPM

As depicted in state of the art, methods that can be found in literature use only virtual instances of the TPM. Even if those methods achieve their goals, they have to change the fundamental principle of trust. Those solutions present two major drawbacks: a software implementation can be affected by defects, and the Verifier should be aware of the communication with a virtual TPM instance.

An excellent example of this application implementation can be found in the article *"vTPM: Virtualizing the Trusted Platform Module"*[20]. In both architecture, the authors introduced provide full TPM functionalities to virtual machines. However, the attestation process is completed by the vTPM, and this latter one applies the final signature. With our implementation instead, the Deep Attestation process is under the full control of an hardware TPM. A binding with a physical TPM guarantee the whole operation process. Not because it relies on other integrity processes but because TPM is considered the most robust root of trust.

4.2.1 Resource allocation

The first phase that is external to the Deep Attestation process is the Resource allocation. During this phase, the VM is allocated, and for this reason, also the necessary resources to support the attestation process later must be provisioned. In the following chapter it will be shown the exact implementation, however for the purpose of the structure it is sufficient to know that the following resources will be provisioned inside the Prover.

Previously, it was introduced the concept of Attestation Data. The measurements collected by the Prover must be aggregated and stored somewhere inside the Physical Platform layer. For this reason the first resources allocated to support this phase in the Deep Attestation process are the *registers*. Those registers, called PCRs, are allocated inside a memory space accessible to only the physical TPM. This strong binding between the VM and the Physical Platform layer makes the Prover to be considered as a single entity for the Verifier. The Prover, on the other hand, needs a pair of keys to be able to subsequently make its digital signature on the attestation data that will be saved on the PCR, to guarantee integrity and non-repudiation to the Verifier. Inside the Prover indeed the VM entity will be bound to a keys pair, later assigned to a certificate.

At manufacture time each TPM is provided with a significant random value, the so-called **primary seed**. All the key generation processes rely on this primary seed, directly or indirectly. Directly when through the TPM is created the primary key. Inderectly when ordinary keys are generated using the Random Number Generator. The hardware component is so able to generate two key pairs: **Endorsement Key (EK)** and **Attestation Identity Key (AIK)**. The **EK** in this architecture plays the role of storage key and is used as a primary parent key to generate new keys pair assigned to different VM instances. AIKs instead are the keys devoted to the signing operations. The latter one has to be bound to every single VM. The aim of using AIK instead of directly using the EK is that this allows the TPM not to expose the EK itself. The hierarchy is presented in the following figure Fig.4.5.

In order to let the Verifier use the public portion of the key and to not overload the usage of the hardware, certificate creations can also rely on Attestation Certificate Authority (ACA). An AIK is indeed a x.509 v3 certificate which contains the public key. This credential is essential in the process because it validates the origin of the AIK used. The only difference between this certificate and the conventional public-key certificates is that the certificate on the AIK also ensures to not reveal the identity of the physical TPM. The whole process will be later expanded in the following chapter, but it is based on the so-called Privacy Certificate Authority schema[21].

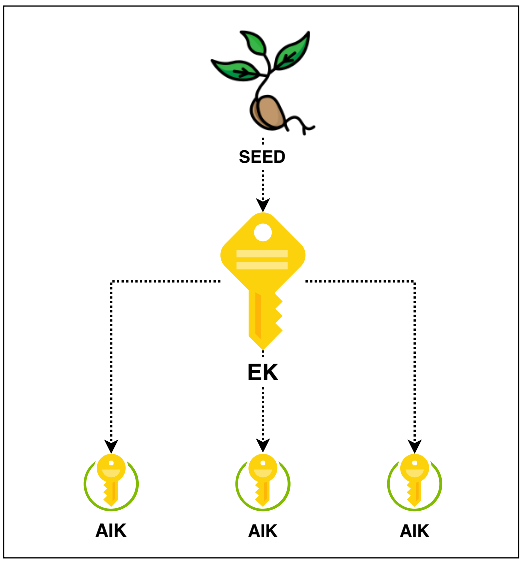


Figure 4.5. Keys Hierarchy

4.2.2 Identity Credential Enrollment

Now that all the necessary resources are allocated, there is one more preliminary phase in the Deep Attestation protocol: the *Identity Credential Enrollment*.

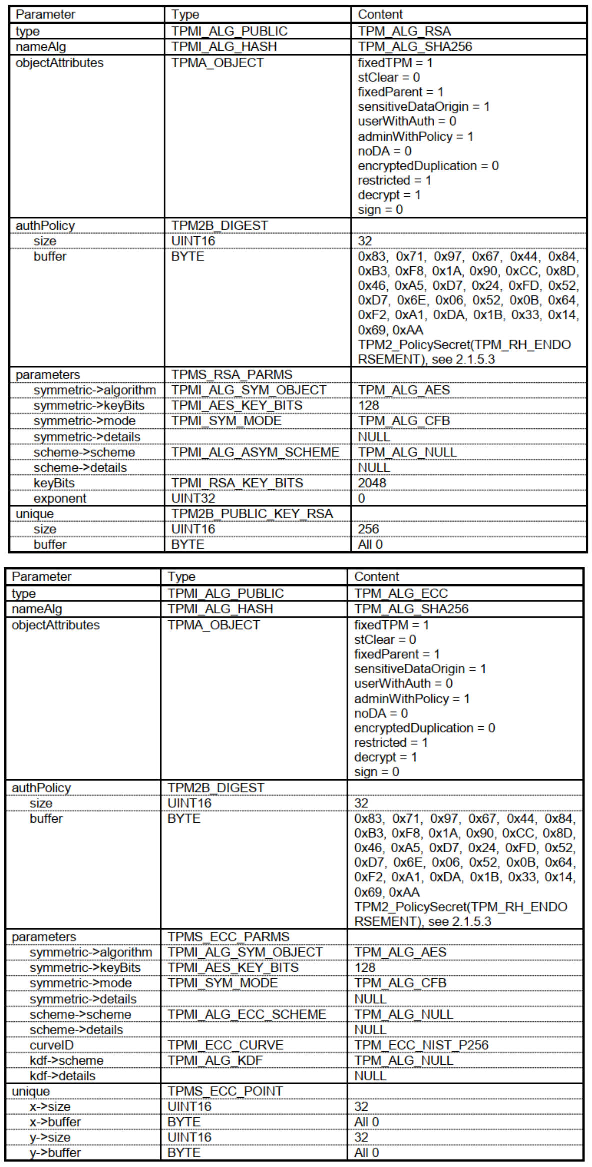
The Prover needs not only a keys pair, the digital signature in order to be validated requires the enrollment of a certificate to be shared with the Verifier. With this procedure the public key is available to the Verifier so that it can correctly evaluate the report integrity and its origin.

The Prover indeed needs to create the certificate through a standardized protocol defined by the ACA. More precisely two different certificates will be issued to the Prover. The first certificate regards the EK that will attest the ownership of the key used for the creation of the AIK by the physical TPM which resides in the Physical Platform of the Prover, this certificate can be used in multiple attestation process for different VM instances. The second certificate is instead issued for the single AIK pairs bound to the VM instance. It is indeed used only the attestation processes that involve the particular VM.

In the following section it will be illustrated the two standard protocols used by remote attestation implementations based on the use of TPM as an attester agent.

Identity credential enrollement - EK

The EK is an asymmetric key pair composed by a private and a public part. The public can be exposed outside, and it is part of the certificate associated to it. The private key, on the other hand, can not be in any way leave the TPM. For the certificate to be issued following the specification for the TPM 2.0 version the EK should be of one those two types: RSA 2048 bit key or ECC NISTP-256 bit key. Below in figure Fig.4.6 the two templates that can be used for the definition of the public portion of the two keys.



Parameter	Type	Content
type	TPMI_ALG_PUBLIC	TPM_ALG_RSA
nameAlg	TPMI_ALG_HASH	TPM_ALG_SHA256

objectAttributes	TPMA_OBJECT	fixedTPM = 1
		stClear = 0
		fixedParent = 1
		sensitiveDataOrigin = 1
		userWithAuth = 0
		adminWithPolicy = 1
		noDA = 0
		encryptedDuplication = 0
		restricted = 1
		decrypt = 1
		sign = 0

authPolicy	TPM2B_DIGEST	
size	UINT16	32
buffer	BYTE	0x83, 0x71, 0x97, 0x67, 0x44, 0x84, 0xB3, 0xF8, 0x1A, 0x90, 0xCC, 0x8D, 0x46, 0xA5, 0xD7, 0x24, 0xFD, 0x52, 0xD7, 0x6E, 0x06, 0x52, 0x0B, 0x64, 0xF2, 0xA1, 0xDA, 0x1B, 0x33, 0x14, 0x69, 0xAA TPM2_PolicySecret(TPM_RH_ENDO RSEMENT), see 2.1.5.3
parameters	TPMS_RSA_PARMS	
symmetric->algorithm	TPMI_ALG_SYM_OBJECT	TPM_ALG_AES
symmetric->keyBits	TPMI_AES_KEY_BITS	128
symmetric->mode	TPMI_SYM_MODE	TPM_ALG_CFB
symmetric->details		NULL
scheme->scheme	TPMI_ALG_ASYM_SCHEME	TPM_ALG_NULL
scheme->details		NULL
keyBits	TPMI_RSA_KEY_BITS	2048
exponent	UINT32	0
unique	TPM2B_PUBLIC_KEY_RSA	
size	UINT16	256
buffer	BYTE	All 0

Parameter	Type	Content
type	TPMI_ALG_PUBLIC	TPM_ALG_ECC
nameAlg	TPMI_ALG_HASH	TPM_ALG_SHA256

objectAttributes	TPMA_OBJECT	fixedTPM = 1
		stClear = 0
		fixedParent = 1
		sensitiveDataOrigin = 1
		userWithAuth = 0
		adminWithPolicy = 1
		noDA = 0
		encryptedDuplication = 0
		restricted = 1
		decrypt = 1
		sign = 0

authPolicy	TPM2B_DIGEST	
size	UINT16	32
buffer	BYTE	0x83, 0x71, 0x97, 0x67, 0x44, 0x84, 0xB3, 0xF8, 0x1A, 0x90, 0xCC, 0x8D, 0x46, 0xA5, 0xD7, 0x24, 0xFD, 0x52, 0xD7, 0x6E, 0x06, 0x52, 0x0B, 0x64, 0xF2, 0xA1, 0xDA, 0x1B, 0x33, 0x14, 0x69, 0xAA TPM2_PolicySecret(TPM_RH_ENDO RSEMENT), see 2.1.5.3
parameters	TPMS_ECC_PARMS	
symmetric->algorithm	TPMI_ALG_SYM_OBJECT	TPM_ALG_AES
symmetric->keyBits	TPMI_AES_KEY_BITS	128
symmetric->mode	TPMI_SYM_MODE	TPM_ALG_CFB
symmetric->details		NULL
scheme->scheme	TPMI_ALG_ECC_SCHEME	TPM_ALG_NULL
scheme->details		NULL
curveID	TPMI_ECC_CURVE	TPM_ECC_NIST_P256
kdf->scheme	TPMI_ALG_KDF	TPM_ALG_NULL
kdf->details		NULL
unique	TPMS_ECC_POINT	
x->size	UINT16	32
x->buffer	BYTE	All 0
y->size	UINT16	32
y->buffer	BYTE	All 0

Figure 4.6. Certificate templates

The EK can also be provided directly with the TPM by the manufacturer, but in both cases, once the key pairs is generated, there is the need for the Endorsement Key Credential. The EK credential is an X.509 v3 certificate that inside contains the public portion of the key pairs created. Inside the certificate, there is also some information regarding the security quality of the physical TPM. In general, the EK credential are an asset to ensure the private EK hold by the TPM is conform to the Trusted Computing Specification. To meet the requirements, the credential issued for the TPM should contain the Public EK, the TPM version, manufacturer, part number and firmware version.

The type of certificate and version is chosen for cross-compatibility with other PKI services. However, some of the fields present in the certificate are specific for a TCG interpretation. There are in particular several proprietary attributes inserted in the subject alternative name or directory attribute extension, that are principally used for those TCG values. Nevertheless, the specification of the certificate follows the profile of the RFC 5280[22] that derives directly by the ITU-T X.509 specifics[23].

Identity credential enrollment - AIK

The EK certificate enrollment is easier since it is bound to the manufacturer that have to attest the compliance status of the TPM to the TCG specifications. The AIK credential instead is issued by and Attestation Certificate Authority. The ACA must be a trusted third party entity capable of validate the EK credential.

The purpose of the AIK certificate is the same as the EK. They have to attest that the public AIK is associated with a valid and compliance TPM. As it was previously described for the EK Credential, the AIK certificate must contain TPM properties, specification conformance and attestation process review. However, for and AIK credential enrollment is crucial the EK certificate. Without it, the Certificate Request can not be completed. In fact, the ACA has the burden of providing validation of the AIK chain, described below:

- Validation of the EK CA certificate
- Validation of the EK certificate
- Proof of Possession (PoP) verification for the EK
- Validation of the platform certificate CA
- Validation of the Platform certificate
- Proof of Possession (PoP) verification for the AIK
- Proof of TPM residence for the AIK

The full enrollment requirements are described in the TCG Certificate Enrollment document[24]. Here below there is a high-level overview of the protocol:

1. The AIK key pairs are created inside the TPM
2. the TSS creates the IDENTITY_PROOF structure that contains the following attributes, used for the certificate creation:
   (a) identityBinding: signature over the IDENTITY_CONTENTS structure (structure defined during the key pair creation)

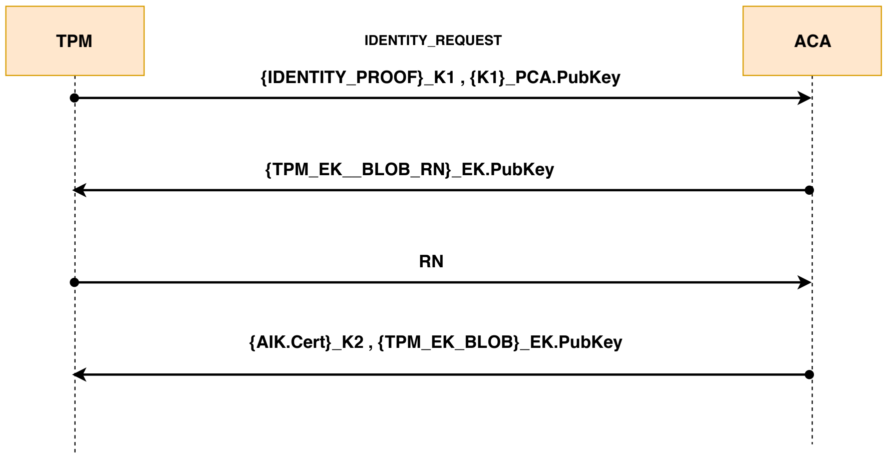

TPM
ACA
IDENTITY_REQUEST
{IDENTITY_PROOF}_K1 , {K1}_PCA.PubKey
{TPM_EK__BLOB_RN}_EK.PubKey
RN
{AIK.Cert}_K2 , {TPM_EK_BLOB}_EK.PubKey

Figure 4.7. AIK certificate enrollment

   (b) TPM version
   (c) AIK public key
   (d) EK certificate
   (e) Platform Certificate

   The IDENTITY_PROOF structure is now encrypted with a symmetric key K1. This symmetric key will be sent encrypted with the public key of the ACA, together with the previous structure.

   The two elements created will be included in a single element called IDENTITY_REQUEST

3. The IDENTITY_REQUEST is sent to the ACA

4. ACA verifies the certificate request received. In particular, it will proceed with the following tasks:

   (a) It will decrypt with its private key the symmetric key K1
   (b) Now that it has access to K1, it can decrypt the IDENTITY_PROOF structure.
   (c) Through all the other elements the ACA is able to recreate the IDENTITY_CONTENTS and verify the signature inside the identityBinding attribute
   (d) the ACA will also validate the EK and Platform certificate

   Even if now the ACA has the whole elements to issue the certificate it has no proof that the AIK private key is inside the TPM and that it has a cryptographic link with the parents key EK.

5. ACA sends a challenge: ACA create the TPM_EK_BLOB_R structure:

   (a) hash of the AIK public key

   (b) random number RN

   The structure is then encrypted with the EK public key present in the certificate received.

6. The structure is sent to the TPM

7. The structure is received and decrypted using the private EK and extrapolate the challenge RN sent by the ACA

8. TPM return the challenge RN to the ACA

9. At this point, the ACA is sure that the TPM has the ownership of AIK key pairs. For this reason, issues a new AIK certificate.

10. ACA encrypts the certificate in a way that is recognizable by the TPM. The structure and the procedure is identical to the challenge phase. A structure called TPM_EK_BLOB is created it contains:

    (a) hash of the AIK public key

    (b) symmetric key K2 used for the encryption of the AIK certificate

    This structure is then encrypted with the EK public key.

11. The certificate is sent to the TPM together with the encryption key K2.

12. Platform is able with all the received information to decrypt the received certificate using the TPM. In particular, the structure received is decrypted using the private EK. From the decrypted structure, the symmetric key K2 is extracted. The key K2 can be now used to decrypt the certificate received.

4.2.3 Platform Integrity Measurement

The process internal to the VM or VMM that is responsible for the integrity measurements is called **Platform Integrity Measurements**. Furthermore, it is the service that interacts with the pTPM. During this phase of the Deep Attestation process the Prover must collect measurements regarding its internal state. The Trusted Platform module is, in fact, in the current scenario the root of trust of measurements, allowing only the root of trust of measurements to write the platform configuration registers.

The Prover must provide evidence of both its boot process both its run-time behavior, and in order to achieve a collection of comprehensive information about the system it uses two tools: Static and Dynamic Measurements.

Static Integrity Measurement

Static measurements take place during the VM system boot and reload. It is based on an immutable piece of code called Core Root of Trust for Measurement. On the common platform, the **CRTM** is usually represented by the firmware, which is the fundamental **Trusted Building Block (TBB)** that remains unchanged during the lifecycle of the VM. Following a bottom-up approach and using a process called Transitive Trust [25] the whole system is measured. **TT** is a process that measures trust at each of the different levels in the system. Fig. 4.8 shows the process.

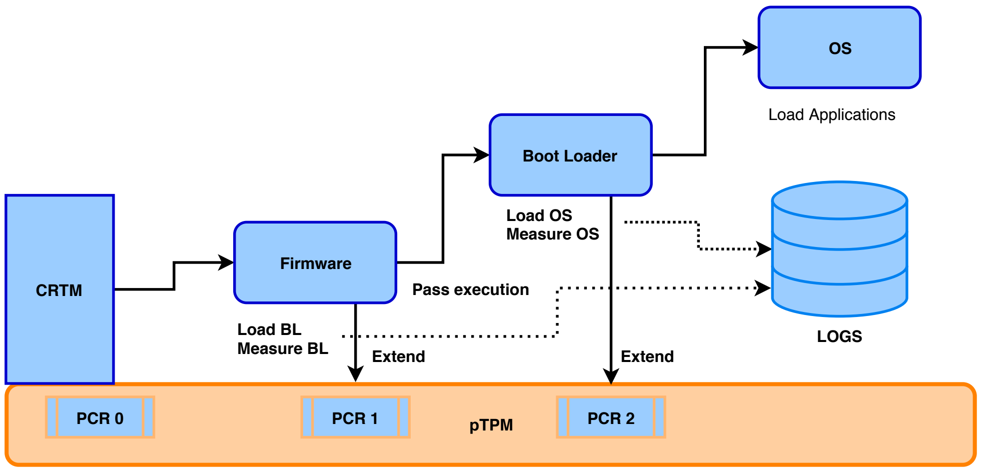


Figure 4.8. Static Integrity Measurement process

When the system is turned on the firmware value, that is acting as **CRTM**, is stored inside the hardware TPM. At this point of the execution, it is responsible for loading the Boot Loader, measure it and save the measures inside the physical TPM. Once the measures are generated and the related logs stored, the BL take the execution control. The Boot Loader now is responsible for loading the OS and follow the same process of the firmware.

The three phases of loading, measure and pass the execution are repeated until the VM is operational and ready to load the application. From this point now the measurements of the system will be taken at run-time.

Dynamic Integrity Measurement

Dynamic Integrity Measurement is again the responsibility of the Platform Integrity Measurements process. Those measurements will be part of the reporting to produce a reliable proof of trust. In fact as reported in the following article [26], the integrity platform definition do not only include essential measures of the machine at boot-time. It has also to include vital measures for all the processes inside a machine at running time.

It is not relevant if the application or the process, in general, is loaded by the OS itself, that was previously attested, or by another application. For each of those processes in which is possible to define a well defined semantic integrity, a similar approach to the one adopted in the static measurements can be adopted. Nevertheless, there are processes in which the integrity state depends on the integrity of other methods. Data collected during the process is later aggregated and saved inside the TPM with the same approach that will be later expanded.

4.2.4 Integrity Request

The Verifier wants to initiate the process of Deep Attestation and for this reason, requests the Attestation Data in which he is interested. The Attestation Data requested corresponds to a set of logs and its aggregate representation inside the registers implemented in the hardware TPM. Those elements will reflect the internal state of the VM. However, in this integrity request, it is indirectly requested the VMM attestation to complete the Deep Attestation process. If the two attestation processes are disjointed could be that in the time between one report and the other one the state of one of the two could have changed. For the Verifier is indeed important to receive the entire Prover state once the Integrity Request is sent. Inside the Integrity Request we find the below attributes:

- **Nonce**. For the freshness of the information
- **Attestation Data request**

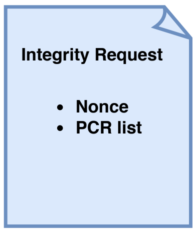


Figure 4.9. Integrity Request

4.2.5 Integrity Report

The Prover that received the Integrity Request by the Verifier now is responsible to collect the measurements and create a compatible structure for the report. The Integrity Report will be composed by the following elements, to which we previously referred as Attestation Data.

- **VM Attested Structure**. The main target of the Attestation process initialized by the Verifier against the Prover is the VM. For this reason The aggregate results of the integrity measurements, stored inside the registers, is collected in this structure.

- **VM Digital Signature**. To prove the integrity and the non-repudation against the previous Attested Structure the Prover must sign with the Attestation Identity Key the structure.
- **VMM Attested Structure**. As it was previously mentioned, the Deep Attestation process must include in its implementation the an integrity proof of all the Prover layers. The structure is in the form the same as the VM Attested Structure and it contains the aggregation of the VMM measurements.
- **VMM Digital Signature**. In the same way as the Verifier needs to test the integrity and the non-repudiation of the VM Attested structure, also in this case its presented to the Verifier a method to attest the truthfulness of the collected measurements.
- **VM Logs**. The previous structures are the method to support the integrity of the collected measurements that are stored inside the measuremts logs. Logs represent the state of the machine in all its life-cycle and with those elements the Verifier has the possibility to support its decision-making process regarding the integrity of the endpoint.
- **VMM Logs**. To have a Deep Attestation the full Prover integrity must be provided to the Verifier.

Once that the attestation process is completed the full Integrity Report can be assembled and sent back to the Verifier together with the Stored Measurement Log (SML) for both the VM and VMM. The SML is naturally stored outside the TPM, but they reflect all the operations that changed the registers values. Those measurements are essential to the Verifier to complete the Report Evaluation process.

4.2.6 Integrity Evaluation

The Verifier now that have received the Integrity Report can proceed with the evaluation of it. For this reason, it will proceed to verify first of all the integrity of the Attested Structure checking the Digital Signature attached in the Integrity Report.

Once that the integrity of the Attested Structure is verified the Verifier can proceed to a double evaluation process. It has the task of comparing the measurements made on the Prover with the know fingerprints that represent the expected state of the system. During this phase, however, it must also check the integrity of the logs that have been collected by comparing them with their aggregate representation within the Attested Structure to check that they have not been tampered.

Once the evaluation is completed, at the VM will be assigned a state of TRUST or UNTRUST. The same process is than repeated for the VMM, whose Integrity Report is inside the report received for the VM.

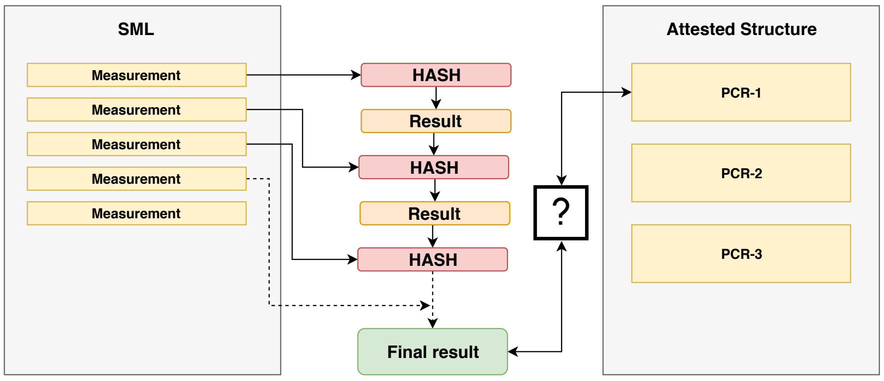


Figure 4.10. Report evaluation flow

Chapter 5

Implementation

As it was introduced in the previous chapter, the core part of the thesis was focused on the introduction of commands to support the Deep Attestation process. More precisely, those commands are crucial to integrate the VM life-cycle and the Integrity Request received by a Verifier. For this reason, the first section will analyze more precisely which commands are introduced, how they support the remote attestation process and more importantly, how they achieve the same security of already existing commands. In the second section, instead, are introduced some structure and classes created in the project to support the demonstration and operation of the added commands. The crucial operations of the attestation process have been recreated. A CA has therefore been introduced with the task of providing the Verifier with the right public key for verifying the digital signature. A Verifier was also created for the creation of Integrity Request and to verify the latter later.

5.1 Changes to pTPM interface

Majority of changes were made to support the life-cycle of a VM and a transparent attestation process from a Verifier point of view. During the creation of a VM, in fact, the TPM is in charge of allocating the right resources for future measurements. Those measurements will be both static and dynamic, depending on needs and implementation. The allocation of resources includes, in particular, the definition in the NV RAM of the registers. Subsequently, these registers will have to be accessible, and a data structure has been introduced, external to the TPM, able to memorize the association between VMs and memory indexes.

Following the allocation of the necessary resources, the environment is ready to receive the necessary measures for a future certification process. The measurements will be saved in the form of logs externally, so then to be sent to the Verifier. However, the digests that represent the state of the machine are saved in the relative indexes allocated before. In this way, it is possible to recognize if the measurements should be modified.

Now that the VM has been allocated along with all its resources and its status is monitored, the Verifier can advance the machine integrity request. After the evaluation of this, it will generate a verdict and place the machine in one of the following states: *trusted, untrusted or unknown.* The request for integrity takes place in a completely transparent manner. The Verifier must be aware of interfacing with a virtual environment because this involves also having to check the layer below the VM. However, it will be able to start the integrity process without further changes to its interface. The binding created, in fact, between the vTPM and the pTPM and the command introduced, keep the creation of an attestation based on the operation of Quote identical to the one that would be obtained if the VM had dedicated hardware. Naturally with the same security as all operations are performed within the pTPM. This procedure, of course, introduces the bottleneck of the implementation and will be discussed in later chapters. However, we need to find a trade-off between wanting a low-cost, faster but software implementation, with all the vulnerabilities involved, and an implementation supported by the hardware that since it has to manage more than a single instance of virtual TPM, it will have some lower reaction times.

At this point it is important to introduce the memory space in which all the VM registers will be allocated, along with a summary of their life-cycle that will be explained in details in the following subsections.

NV-RAM

TPM has a limited amount of PCRs, and a one-to-one mapping with the virtual PCRs is not feasible. For this reason, the storage of the virtual-PCR should be done in a different memory location.

The non-volatile RAM storage is a memory location in which is implemented a restricted access control. In general, it has multiple implementations. Inside it, it is possible to store keys and provide a mechanism for faster access to data. The possibility of control read and write capabilities represents the most crucial peculiarity. In particular, having access to an NVRAM provides you with the following capabilities:[1]

- Storage of root keys used in the certificate chain
- Storage of endorsement key
- Storage of representative machine states
- Storage for decryption keys

Because of all its functionalities and the restricted-access control is the perfect place to emulate and define inside this memory location the virtual PCR instances. Saving those PCR inside the NVRAM that can be modified and accessed by only the TPM, ensures the perfect environment for a proper replication of the physical PCR behaviour. PCRs that are responsible for the storage of measurements taken during the Platform Integrity Measurements are now mapped inside the NVRAM.

Every time that a VM is allocated, after the creation of the key for the attestation, its set of PCRs are allocated inside the NVRAM. Each non-volatile PCR has a reference index that is consequently bound to the VM through a mapping for faster research.

PCR life-cycle management

Before the first booting of the virtual machine, the PCRs set inside the NVRAM should be created. Once the necessary registers and bank are deployed, it is possible to involve them in essential operations. PCRs are primarily engaged during the Integrity Measurement to store the appropriate data inside the registers. Secondly, they are involved during the Integrity Reporting phase because they have to take part in the core operation of the Deep Attestation protocol: Quote operation. The overall lifecycle is represented in the following Fig.5.1:

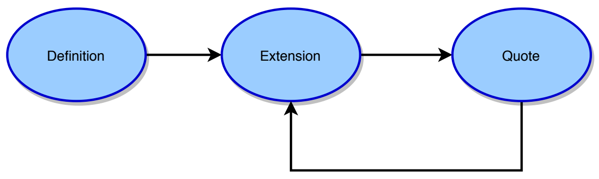


Figure 5.1. PCR lifecycle management

5.1.1 Resource allocation - nvPCR definition

The first phase, as previously introduced, is the allocation of resources. In particular, the VM will need a set of registers that will simulate the physical PCRs usually present in a TPM hardware.

The memory allocation process is mainly supported by the following main functions:

- `Allocate_VM_Resoruces`
- A new ESAPI
- `Define_nvPCR`
- `UpdateMap`, a support function

Allocate_VM_Resources

`Allocate_VM_Resources` is the primary function that has the task of managing the entire memory allocation process. The inputs of this function is the unique identifier of the virtual machine and a variable representing the first memory index that can be allocated: `vID`, `freeIndex`. A logical implementation of the function is depicted in the algorithm 1.

Algorithm 1 Algorithm for resource allocation

```
 1: function ALLOCATE_VM_RESOURCES($vID$,$lastIndexDefined$)
 2:     $freeIndex$
 3:     $nvHandle$
 4:     $i \leftarrow 0$
 5:     $hashBank1 \leftarrow TPM2\_ALG\_SHA1$
 6:     $hashBank256 \leftarrow TPM2\_ALG\_SHA256$
 7:     for $i < 23$ do
 8:         $freeIndex \leftarrow$ FirstFreeIndex()
 9:         $nvHandle \leftarrow$ Define_nvPCR($hashBank1$,$freeIndex$)
10:         UpdateMap($vID$,$hashBank1$,$i$,$nvHandle$)
11:     end for
12:     $i \leftarrow 0$
13:     for $i < 23$ do
14:         $freeIndex \leftarrow$ FirstFreeIndex()
15:         $nvHandle \leftarrow$ Define_PCR($hashBank256$,$freeIndex$)
16:         UpdateMap($vID$,$hashBank1$,$i$,$nvHandle$)
17:     end for
18: end function
```

When a virtual machine is added to the deployment, two banks of registers are allocated for it. The register banks are characterized by the type of digest that can contain the registers inside them. In this circumstance, sha1 and sha256. The resource allocation process must take into account and update every time a new memory space is allocated, the following variables:

- `freeIndex`: to keep the pointer up to the first available memory index
- `iterator`: it also logically represents the PCR number.

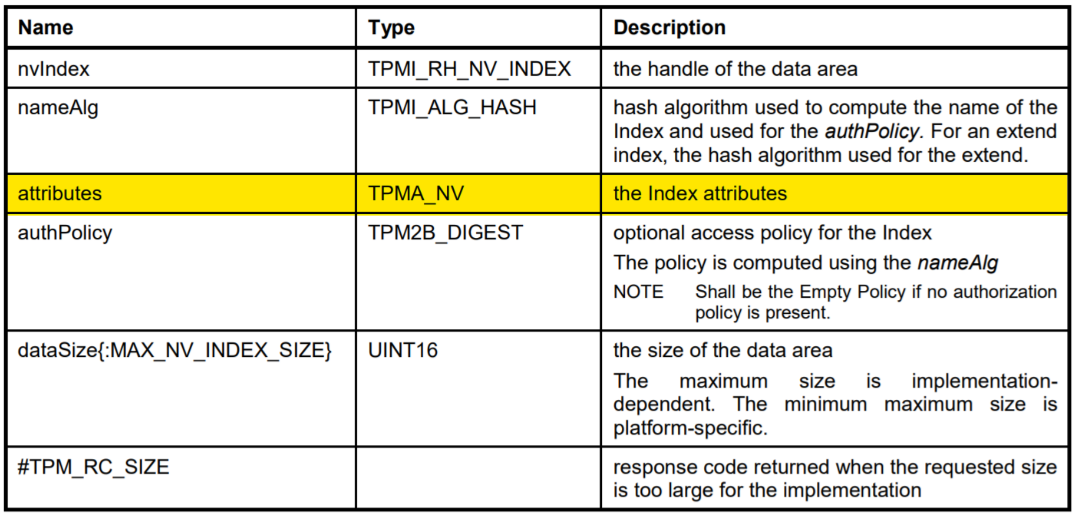
Furthermore, the structure that contains the mapping between the virtual machine identifier, the bank (represented by the hash algorithm), the PCR number and the index identification handle must be updated. The latter will then be necessary in order to access the right register later.

It is essential to develop a function for the research of the first available index. In fact, even if logically the registers are contiguous, in the memory space NV they may not be. This happens when, for example, the resources of a particular virtual machine are released because it has been removed or moved.

Define_nvPCR

The Define_nvPCR function is instead the first change to the pTPM interface. It is the definition of a new Enhanced System API. It is useful for automating the creation of memory spaces with particular attributes to simulate a PCR register within the NV memory correctly.

This API, in fact, has as its central point the definition of a memory space that simulates the PCR but in the NV memory. In the Fig.5.2, it is possible to have a view of the parameters that can be set according to the TCG specifications[27][28].



Name	Type	Description
nvIndex	TPMI_RH_NV_INDEX	the handle of the data area
nameAlg	TPMI_ALG_HASH	hash algorithm used to compute the name of the Index and used for the *authPolicy*. For an extend index, the hash algorithm used for the extend.
attributes	TPMA_NV	the Index attributes
authPolicy	TPM2B_DIGEST	optional access policy for the Index The policy is computed using the *nameAlg* NOTE Shall be the Empty Policy if no authorization policy is present.
dataSize{:MAX_NV_INDEX_SIZE}	UINT16	the size of the data area The maximum size is implementation-dependent. The minimum maximum size is platform-specific.
#TPM_RC_SIZE		response code returned when the requested size is too large for the implementation

Figure 5.2. Definition of TPMS_NV_PUBLIC Structure

Of particular interest are the values that can be set as attributes. This structure allows to keep track of data and set restrictions on who can manipulate memory space. In particular, the bit corresponding to the attribute TPM2_NT must be set, which defines the type of index in our case, it must be of type Extend (TPM2_NT_EXTEND). The other parameters instead depend on when the function is called. As in the case of the nvIndex, which will correspond to the first index available in our memory space. Alternatively, to nameAlg, which instead depends on which memory bank we are allocating. A logical implementation of the API is depicted in the algorithm 2.

Algorithm 2 Logical implementation of Define_nvPCR API

```
1: function DEFINE_NVPCR($hashAlg$,$freeIndex$)
2:     $sizeOfHashx$
3:     $nvPublic$
4:     $freeIndex \leftarrow freeIndex$
5:     $authValue \leftarrow TPM2B\_AUTH$
6:     Setting the $nvPublic$ structure
7:     Definition of the space
8:     if Error then
9:         return EXIT_FAILURE
10:    else
11:        return $nvHandle$
12:    end if
13: end function
```

UpdateMap

For the virtual PCR management is necessary to store the binding between the index, that is the reference of the NV location, and the corresponding Virtual Machine that is the owner of this particular index. However, PCRs are also subdivided into banks. In fact, the size of the value that can be stored inside a TPM is determined by the size of the digest generated by the chosen hash algorithm. The minimum requirement from the TCG Specification is to include at least one PCR with 23 registers[29]. The information that has to be stored are the following.

- **VM identifier**: unique number that identifies the VM
- **PCR number**: the register number
- **Bank**: that is identified by the hash algorithm used.
- **NV-Index**: the corresponding index associated with the location memory.

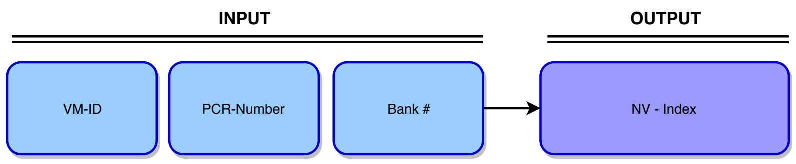


Figure 5.3. Multimap structure

As can be seen from the Fig.5.3 representing the multimap structure, the first three values are input to retrieve the corresponding NV-index. An NV-index is a space that will be defined during the deployment of the virtual machine. A unique handle value identifies it. More specifically, the NV index structure is the following:

- **Unique handle**. It is used to retrieve the reference Index
- **nameAlg**. The hash algorithm used in the computation of the Name of the index.
- **Authorization policy**. It is an optional parameter that represents the digest of the applied policy.
- **Index attributes**. They determines the nature of the index.
- **Authorization value**.
- **Size of the index data**. It represents the number of octets used to hold the NV data
- **NV index data**

The Name of the index is produced using the public portion of the NV index. All the previous arguments compose it except for the Index Data and the authorization value. The **Index Name** is then produced hashing this public portion. That information are usually stored outside the NVRAM to save space for more PCR definition. Nevertheless, the data stored inside this multi-map structure can be considered sensitive. The structure can be saved in the standard storage or the TPM allow the definition of a secure memory location, whose access is controlled by the TPM. Both solution are eligible and depends on the level of security you want to achieve. The use of external NV is allowed by the TCG specification[2]. The only requirement is the application of algorithms with an higher security strength of any other algorithm executed in the TPM for the encryption, integrity check and rollback protection. Other specifications regard the encryption key that should be protected with a level of security as strong as the keys minimum. Moreover, those keys can not be stored outside the TPM even if encrypted, and they must be derived from a seed that does not move from the TPM.

In the implementation the `updateMap`, for a more straightforward and efficient implementation, was chosen a Hash Table.

5.1.2 Measurements aggregation - Extend operation

The measures, as described above, are of two types: *static* and *dynamic*. Static measurements are collected during the boot process. The Dynamic ones, on the other hand, are taken at run time. In both examples, all the measurements will be taken and inserted inside their respective log files. However, the system will maintain an aggregate of integrity measurements over one of the PCR.

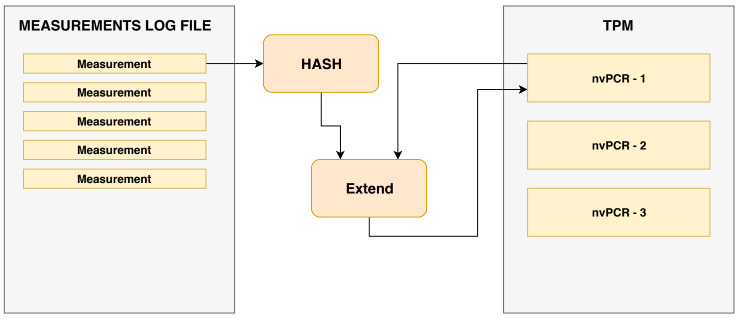


Figure 5.4. Extend process

Since we have defined PCRs as extend types, the process to update them is the `TPM2_NV_extend` command. Also, in this case, a new function has been introduced that can automate the process, and that takes into consideration the data structure from which it is possible to extract the information necessary to update the right PCR. The function takes four values as input:

- `vID`: it will identify the respective VM.
- `PCR number`: accordingly to the measurement taken, the aggregate of integrity will be stored in different registers.
- `hashAlg`: this value identify the correct register bank.
- `data`: it corresponds to the measurement taken.

Through those input the correct registers will be uploaded.

Algorithm 3 Logical implementation of Extend_nvPCR

```
1:  function EXTEND_NVPCR($vID$, $pcr$, $hashAlg$, $data$)
2:      $nvHandle \leftarrow$ SearchInMap($vID$, $pcr$, $hashAlg$)
3:      $dataToExtend \leftarrow$ Marshal TPM2B_MAX_NV_BUFFER ($data$)
4:      Esys_NV_Extend($dataToExtend$, $nvHandle$)
5:      if Error then
6:          return EXIT_FAILURE
7:      else
8:          return $nvHandle$
9:      end if
10: end function
```

As can be deduced from the logical implementation represented by the algorithm 3, first of all, we need to extract from the Hash Table the *handle* that represents the memory space in which the PCR resides. Once the handle has been extrapolated from the HashTable, it is possible to update the register value. The nvPCR will be updated naturally with the result of the hash function on the concatenation of the value already present in the register and the new value.

$$nvPCR \rightarrow newValue = Hash_{hashAlg}(nvPCR \rightarrow oldValue||data) \tag{5.1}$$

A particular point in the implementation is the PCR number 0. The PCR 0 is usually chosen as CRTM. The computer reset puts the system in a known state because, at this point, the processor begins executing the booting process. The initial code is the core of CRTM. The trust of code relies on the manufacturer. Usually, the action of the CRTM is to extend a PCR with a value representative of the CRTM. Some system naturally implement a different method for starting the chain of trust. In those cases, usually, the CPU is considered the CRTM. The peculiarity of TPM is that it supports the hardware-based core root of trust for measurement: H-CRTM. Particular interface indications allow the TPM to understand if it is receiving data from the H-CRTM.

5.1.3 Attestation function - Quote

Once that the environment is ready and functional, the host or client is now ready to authenticate its hardware and software to a remote server or Verifier. Remote Attestation aim is indeed to enable the remote Verifier to verify the level of trust of the platform. The level of trust is based on the integrity of the platform. In the scenario proposed, the action of attestation consists in having the TPM sign the internal PCR values. In a standard scenario, this is achieved using usual attestation protocols that are based on the `TPM2_QUOTE` command. The command produces a regular attestation structure. The block is then hashed and signed by the proper signing key using a chosen signing scheme. The choice of the signing scheme is mainly based on the signing key used for the attestation process.

Naturally, the process has its drawbacks[30]. The Verifier or challenger, in fact, is in charge of make a judgment regarding the software that is running in the machine. The trust decision is usually based on whitelists. However, maintaining a database of trust measurement is not a simple task. According to the application and the white list used for the measurement during the previous phase, TPM should attest only the requested data. Each virtual machine will, in fact, use different paradigms as regards the boot process, and therefore static measures, and different measurement architectures, as far as run-time measurements are concerned (an example of the latter is IMA). Each paradigm or architecture will use different PCRs in different banks for different purposes. At a TPM level, this is not very relevant as the TPM itself is not able to judge the state of the machine but only to securely aggregate the measurement logs. Despite this, it is important that as an input to the attestation process the TPM receives the list of PCRs of interest to the Verifier.

Taking as a reference Fig.4.1 in **Section 4**, we are in the phase in which the Verifier sends its Integrity Request composed of:

- `Nonce`
- `PCR_List`

At this point, the agent will be responsible for retrieving the necessary information and supplying it to the Verifier. The creation of the integrity report can, in fact, be divided into two phases.

- **Reading PCR**
- **Attesting PCR**

An API was therefore created for the management of resources and TPM and to automate the entire process. The whole process is depicted in the logical representation of it: Alg.4. After that, all the steps will be explained in detailed.

Reading PCR

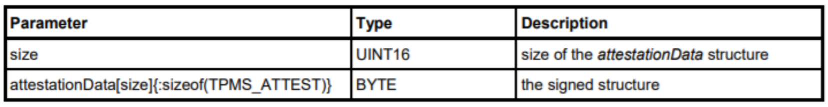
As previously mentioned, the PCR certification can be divided into two: **read** the PCR and **certify** the PCR. The function has as input the registers to be attested. In order to read the correct registers, it is first necessary to retrieve the corresponding handles. The latter are saved within the HashTable data structure (Point 1 in the algorithm).

Once retrieved the handler, it is possible to access the right memory spaces inside the memory and read the value of the registers. At this point, two different operations will be performed:

- PCRs will be saved in a structure that will be included in the Integrity Report. From TPM2.0 version, in fact, the actual PCR values must be provided separately and are not included inside the attested structure [2].
- PCRs are chained in a single buffer to be hashed later. This value is the one that will be included in the attested structure.

Attesting PCR

Once the access to the registers is provided, the registers reading phase is complete, and it is possible to proceed with the creation of the attestation structure.



Parameter	Type	Description
size	UINT16	size of the *attestationData* structure
attestationData[size]{:sizeof(TPMS_ATTEST)}	BYTE	the signed structure

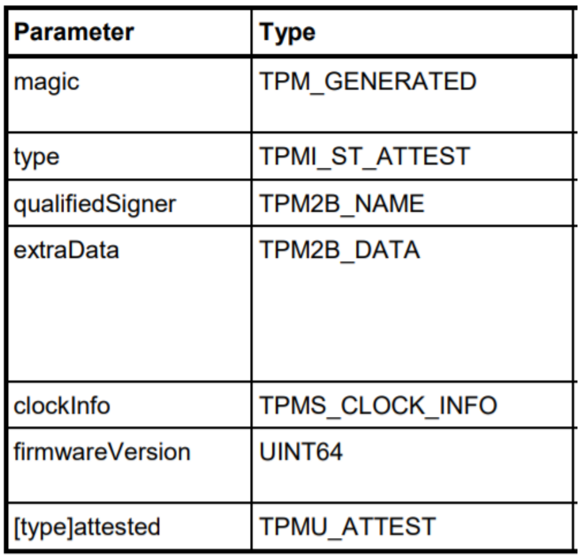
Figure 5.5. Attestation structure

Following the specification of the TCG, the attested structure must reflect the Fig.5.5. This structure will be part of the Integrity Report sent to the Verifier and to be fully transparent to it, each field must be able to be recognized by the Verifier. The `attestationData` is the only signed part of the structure. Indeed the size parameter is not part of the signature. For this reason, it is essential to create first of all the attestation data structure of type `TPMS_ATTEST` Fig. 5.6.

- `magic`. The first field is the called `magic`, this value is used to prevent the TPM to compute digest, and later sign them, received from outside that starts with this particular value. A TPM-generated message always starts with this particular digest. This particular internal protocol ensures that an attacker does not forge the message.

Algorithm 4 Logical implementation of Virtual quote operation

```
 1: function TPM2_VIRT_QUOTE($vID$, $pcrSelection$, $Nonce$, $signScheme$,
    $pubEK$, $pubAIK$, $sigScheme$)
 2:     $attestedStructure$ initialization
 3:     $signature$
 4:     $pcrValues$
 5:     Load_Key($pubEK$, $pubAIK$ )    ▷ Loading the signing AIK key in the TPM
 6:     ▷ Extracting nvHandles (1)
 7:     for $i < pcrSelection.cout$ do
 8:         $hashAlg \leftarrow pcrSelection[i].hash$
 9:         $sizeOfConcatenation +=$ SizeOfDigest($hashAlg$)
10:         $pcrNumber \leftarrow$ BitMapExtrapolation($pcrSelection[i]$)
11:         $nvHandle \leftarrow$ SearchInMap($vID$, $pcrNumber$, $hashAlg$)
12:         $arrayNvHandle[i] \leftarrow nvHandle$
13:     end for
14:     ▷ Reading the selected nvPCR and chaining the results
15:     $pcrConcatenation = \{.size \leftarrow sizeOfConcatenation, .buffer = \{\}\}$
16:     for $i < pcrSelection.cout$ do
17:         $pcrConcatenation.buffer \leftarrow$ NV_Read($arrayNvHandle[i]$)
18:     end for
19:     ▷ Creation of the attestedStructure
20:     $magic \leftarrow$ TPM_GENERATED
21:     $attestedStructure \leftarrow$ Marshal_AttestesStructure($magic$)
22:     $type \leftarrow$ TPMI_ST_ATTEST_QUOTE
23:     $attestedStructure \leftarrow$ Marshal_AttestesStructure($type$)
24:     $qualifiedSigner \leftarrow$ ReadPublic($pubEK$)
25:     $attestedStructure \leftarrow$ Marshal_AttestesStructure($qualifiedSigner$)
26:     $extraData \leftarrow Nonce$
27:     $attestedStructure \leftarrow$ Marshal_AttestesStructure($extraData$)
28:     $clockInfo \leftarrow$ Read_Clock()
29:     $attestedStructure \leftarrow$ Marshal_AttestesStructure($clockInfo$)
30:     $firmwareVersion \leftarrow firmware$
31:     $attestedStructure \leftarrow$ Marshal_AttestesStructure($firmwareVersion$)
32:     $outHash \leftarrow$ Hash($pcrConcatenation$)
33:     $attested.pcrSelect \leftarrow pcrSelection$
34:     $attested.pcrDigest \leftarrow outHash$
35:     $attestedStructure \leftarrow$ Marshal_AttestesStructure($attested$)
36:     ▷ Now the structure can be signed
37:     $quoteHash \leftarrow$ Hash($attestedStructure$)
38:     $quoteHashTicket$                        ▷ produced by the previous Hash
39:     $signature \leftarrow$ Sign_AIK($quoteHash$, $quoteHashTicket$)
40:     return $signature$, $attestedStrucuture$, $pcrValues$
41: end function
```



Parameter	Type
magic	TPM_GENERATED
type	TPMI_ST_ATTEST
qualifiedSigner	TPM2B_NAME
extraData	TPM2B_DATA
clockInfo	TPMS_CLOCK_INFO
firmwareVersion	UINT64
[type]attested	TPMU_ATTEST

Figure 5.6. Signed structure

- `type`. Since we are performing an attestation based on a quote operation, the next constant field type must be set to `TPM_ST_ATTEST_QUOTE`. This field instead define from which operation the attested structure was created. The attested structure is in fact also created for Certify operation or Audit session, for example.

- `qualifiedSigner`. It is the Qualified Name of the key used to sign the attestation data. This QN is useful to the Verifier to determine where the signature is produced.

- `extraData`. It is the field in which the TPM should put the Nonce received by the Verifier. In this way, the TPM is preventing the replay attacks.

- `clockInfo`.It is not a single parameter, but it is composed of 4 of them. The values included in this structure are useful to have more information about the status of the TPM. In particular, a reference when the TPM was last switched on, number of occurrences of TPM reset, number of shutdown or start.

- `firmwareVersion`. It is a vendor-specific value and identifies the version number of the firmware.

- `attested`. Here instead we have the type-specific attested information that in the scenario are the Quote information: `TPMS_QUOTE_INFO`. This structure contains the two primary information. This information regards algorithms and PCR selected and the actual digest of the selected PCR using the hash of the signing key. As can be seen from the algorithm; once the PCR values have been read, they are concatenated in a single buffer. From this buffer, a digest is produced and subsequently inserted into the structure. The hash operation is performed using one of the symmetric primitives included in the TPM.

The attested structure is now completed. Using the primitive Hash function again, included inside the TPM is possible to create the digest used later for the sign operation. Also, in this case, the Hash function will produce two outputs:

- **Digest**
- **Validation(Ticket)**

The *ticket* is significant during this operation. The hash will be used in a signing operation, and the *ticket* represents that the digest operation happened inside the TPM, and it is safe to sign the digest. This particular operation prevent the TPM to perform involuntary a digital signature.

To perform the signature over the attested data structure is used again directly the TPM. The private AIK is loaded through its public key and used for the signature. Finished the signature phase, the TPM will return the following parameters that will compose later the final Integrity Report.

- `signature`
- `attestedStructure`
- `pcrValues`

Now only half of the deep attestation process is completed. To create the **Integrity Report** the Agent will perform a standard attestation process over the VMM to retrieve the integrity status of the layer under the VM. Once the **Integrity Measurements** of the VMM have been retrieved, the Integrity Report can be composed. The Verifier, at this point, has to perform a double verification. First, the content of the structure that contains the PCRs must be confirmed and with the fact that it was signed by TPM. For this reason, it will recalculate the PCR digest obtained and compare it with the one signed by the TPM. If the two values match, it can turn to verifying the integrity of the VM and the VMM. It can trace the status of the registers by following the operations that were saved in the received logs. Once finished, it compares its results with the ones collected in the PCRs. Based on the outcome of this comparison, the Verifier can verify whether the attested machine can be considered trust or untrust. The value of the measurements in the log files, in fact, can be altered but their aggregate value in the form of digest is not modifiable.

5.2 Proof of Concept

In this section, it will be possible to see the actual implementation of the remote attestation process. For the simulation, different *c++* classes will represent the various entities in the Deep Attestation Process. Two separate attestation will be performed. In the first attestation, the Logs inside the Prover will not have been tampered, and the final evaluation of the VM will mark the machine as TRUST. In another remote attestation process instead, the VM will be marked as UNTRUST since the results of the Report Evaluation will show that the integrity of the machine can not be confirmed due to a discrepancy between known fingerprints and reported logs.

For a better understanding of the workflow and for an adequate understanding of how the various components involved in the remote authentication process have been implemented, in the following chapter will explain in detail the developed classes and the data structures involved.

5.2.1 Classes and data structures

In order to simulate the remote attestation process but with particular attention to those processes at a lower level that involves the TPM directly some higher level and more cosmetic functions have only been partially implemented. In this way, it was possible to emphasize the internal interaction of the various components.

The remote attestation process, as previously described in section x, involves the interaction between the following components: the Verifier, the Prover and the Attestation certificate authority. During the process, the components exchange the following structures: Integrity Report, Integrity Request and the related public keys linked to the private keys used for the digital signatures made on the reports.

Prover

The Prover is the focal point of the remote attestation and includes several components within it. They are in particular responsible for the Platform Integrity Measurement and the Integrity Reporting.

Within the thesis project, particular attention was given to the figure of the Prover as changes were made to the internal design of some components to achieve the final objective. For this reason, it was preferred not to represent it as a single class but to divide it into its two main components: the pTPM and the vTPM. For this reason, two classes have been created. The reason for this choice is mainly due to the demonstration that it is possible to use within a prover a single physical component capable of managing more than one instance of vTPM.

The methods and variables that make up the pTPM class, functional to the deep attestation flow, can be divided into two: public and private. Public functions are implemented to ensure correct communication with the other components.

```
class pTPM
{
public:
```

```
// Constructor
pTPM(int number);
// Destructor
~pTPM();
// Function for Resource Allocation
void Allocate_VM_Resources(UINT32 vID);
// Function for Resource release
void Release_VM_Resources(UINT32 vID);
// Function for the creation of the primary key associated with
   th pTPM
void CreatePK();
// Function for the creation of a derived key associated with a
   vTPM
void Create_Key(Key* derivedKey, Key* primaryKey);
// Function for the creation of a derived key associated with a
   vTPM
void CreateAIK(UINT32 vID);
// Function for the extension of the space
void Extend_nvPCR(UINT32 vID, int pcrNumber, TPMI_ALG_HASH
   hashBank, TPM2B_MAX_NV_BUFFER* dataToExtend);
// Function for Handling the Integrity Request received from
   the vTPM
IntegrityReport IntegrityRequestFrom_vTPM(UINT32 vID,
   IntegrityRequest integrityRequest);
//Function to find the memory allocation of a PCR
ESYS_TR GetIndexFromMap(UINT32 identifier, TPMI_ALG_HASH hash,
   UINT32 pcrNumber);
//Export public Key
Key GetRootKey();
Key GetAIK();
//Export public Key
Key* GetRootKeyP();
Key* GetAIKP();
}
```

In addition to the public methods, also some private functions have been developed, with the task of directly managing internal resources, that needs to be addressed with precise sequences of internal resource calls.

```
class pTPM
{
private:
// Method to initialize the TCTI context
void Init_Tcti_Tabrmd_Context();
// Method to finalize the TCTI context
void Finalize_Tcti_Tabrmd_Context();
// Methods to initialize the ESYS context
void Init_Esys_Context();
// Method to finalize the ESYS context
void Finalize_Esys_Context();
// Error handling function
int ErrorHandling(TSS2_RC rc);
// Function to extract hash size
UINT16 GetHashSize(TPMI_ALG_HASH hash);
// Function for nv-index definition
ESYS_TR Define_nvPCR (TPMI_ALG_HASH hashBank, TPMI_RH_NV_INDEX
   freeIndex);
// Function for undefining an NV space
void Undefine_NV_Space(ESYS_TR* handle);
// Function for Updating the MAP
void UpdateMap(unordered_map<MapKey,ESYS_TR>* mapNVarea, UINT32
   identifier, TPMI_ALG_HASH hash, UINT32 pcrNumber, ESYS_TR
   nvIndex);
// New command for deep attestation
/*
vID [in]
pcrSelection [in]
nonce [in]
inScheme [in]
signingKey [in]
primaryKey [in]
attestStructure [out]
signatureStructure [out]
pcrValues [out]
*/
void TPM2_VIRT_QUOTE(UINT32 vID, TPML_PCR_SELECTION
   pcrSelection, TPM2B_DATA nonce, TPMT_SIG_SCHEME inScheme,
   Key* signingKey, Key* primaryKey, TPM2B_ATTEST*
   attestStructure, TPMT_SIGNATURE* signatureStructure,
   TPML_DIGEST* pcrValues);
}
```

Among the functions and methods implemented, we find the fundamental steps of the remote attestation process. In particular, The resource allocation explained in detail in Section 4.2.1 is implemented through the Allocate_VM_Resource function with which the PCR banks are reserved inside the NVRAM memory. In particular, two banks are reserved, one with indices with Has SHA1 algorithm and one with SHA256. This phase is in fact addressed in the first steps in the `main.cpp`, which has been developed to simulate the workflow of the Remote Attestation process. Within the code, following the classic calls to Constructors, we find precisely the memory allocation of resources useful to prepare the development environment for the proof of concept. For simulation purposes at this point, also the resources devoted to storing the VMM measurements are allocated.

```
// Definition of the NV space of type extend
cout<<"\n***␣Definition␣of␣the␣NV␣space␣␣***"<<endl;
```

```
pTPM.Allocate_VM_Resources(vTPM_1.getIdentifier());
//VMM banks allocation
pTPM.Allocate_VM_Resources(0);
```

As can be seen from the functions developed, the pTPM is identifying each vTPM with a unique number previously assigned. The function internally allocates for each bank type, SHA1 and SHA256, four PCRs. For simulation purposes, this is the number that has been chosen. For the definition of each memory space inside the NVRAM the pTPM class exploits two private functions,`Define_nvPCR` and `UpdateMap`. The first method is clearly used for the definition of the type of PCR, which must be of the Extend type, to follow the rules for defining the functioning of a PCR. The other function is instead use to keep track internally for the binding between the NVRAM and the vTPMs. The actual implementation and choice for the structure is already explained in the previous sections, in particular in the subsection 5.1.1.

In addition to the previous methods, following the natural simulation workflow, we find the functions that are related to the keys management and creation. They are invoked immediately after the resource allocation since they are functional for the correct completion of the process. Those functions, in particular, are:

- **CreatePK**. For the creation of the primary key.
- **Create_Key**. For the creation of the attestation identity key.
- **CreateAIK**. This function is used to invoke the previous function, but it takes as input only the vID that identifies the vTPM associated with the key. In this way, the private key is kept as a private parameter and to invoke the creation of a key it is needed only to pass a parameter that identifies the vTPM.

In fact, for each virtual-TPM, it is necessary to create a key pair in order to sign the Integrity Report generated for a specific VM. The previous functions are in fact invoked exactly adter the resourse allocation, as already mentioned. Below the actual implementation in the `main` program.

```
cout<<"\n***␣Creation␣of␣a␣primary␣Key␣for␣the␣pTPM␣***"<<endl;

//pTPM.Create_PK(&primaryKey);
pTPM.CreatePK();

cout<<"\n***␣Creation␣of␣the␣AIK␣from␣the␣primary␣key␣***"<<endl;

//pTPM.Create_Key(&keyVM1, &primaryKey);
pTPM.CreateAIK(vTPM_1.getIdentifier());
```

In order to properly store the information regarding the keys created, as they include more than one parameters, a structure is developed.

```
struct IntegrityReport
{
TPM2B_ATTEST attestStructure_VM;
TPMT_SIGNATURE signatureStructure_VM;
TPML_DIGEST pcrValues_VM;

TPM2B_ATTEST attestStructure_VMM;
TPMT_SIGNATURE signatureStructure_VMM;
TPML_DIGEST pcrValues_VMM;
};
```

Following the workflow, we find the two main functions implemented to achieve the purpose of remote attestation: `Extend_nvPCR` and `IntegrityRequestFrom_vTPM`. The first function aims to emulate the classic Extend operation that is normally performed on PCRs. In the solution created, the physical TPM manages more than virtual TPM, and for this reason, it keeps the various PCRs relating to the different virtual TPMs in the NVRAM memory. For this reason, through the use of the multimap structure and a unique number assigned to each virtual TPM, the pTPM can trace the memory location of the vPCR in question, more precisely through another support function: `GetIndexFromMap`. Once the vPCR memory location is retrieved, a regular Extend operation is performed on it. The second function in question, `IntegrityRequestFrom_vTPM`, has the purpose of managing the Integrity Report requests coming from the Verifier. The Integrity Request that the pTPM receives has been mapped with the following structure:

```
struct IntegrityRequest
{
TPM2B_DATA nonce;
TPML_PCR_SELECTION pcrSelection;
};
```

Once the parameters necessary to perform the Quote operation, nonce and the list of PCR involved are extracted, a private function is invoked which has the purpose of emulating the Quote operation which is normally carried out with the `TPM2_QUOTE` command. This function is `TPM2_VIRT_QUOTE`, it has the purpose of providing as output the parameters necessary for the construction of the Integrity Report. They include the structure containing the reading of the affected PCR:

- `attestStrucuture`.
- `signatureStructure`.
- `pcrValues`.

The structure created to store the Integrity Report is the following, and it includes naturally not only the Attestation Measurements related to the vTPM but also the integrity information regarding the VMM, which are necessary to complete the Deep Attestation process.

```
struct IntegrityReport
{
TPM2B_ATTEST attestStructure_VM;
TPMT_SIGNATURE signatureStructure_VM;
TPML_DIGEST pcrValues_VM;

TPM2B_ATTEST attestStructure_VMM;
TPMT_SIGNATURE signatureStructure_VMM;
TPML_DIGEST pcrValues_VMM;
};
```

The logic of this function, being the focal point of the thesis project, is explained in the previous section, in particular in the Section 5.1.3, while more information regarding the parameters involved in the composition of the integrity report is explained in Section 4.2.5.

The remaining functions have a more marginal purpose than the remote attestation process and are more involved in the management of the simulation environment. `Init_Tcti_Tabrmd_Context`, `Finalize_Tcti_Tabrmd_Context`, `Init_Esys_Context`, `Finalize_Esys_Context` are used to handle the different simulations contexts. More information regarding those functions are explained in the next section during the workflow explanation. `ErrorHandling` is a function used to handle the different possible error codes received from the TPM commands calls.

The second component that is part of the Prover is the vTPM. Within the simulation, it has a slightly secondary role since all the focus has been placed on the critical part of the project, namely the pTPM. However, the vTPM has the task of interacting directly with the Verifier and using the pTPM to ensure the correct performance of all operations aimed at the remote attestation process.

```
class vTPM
{
[...]
private:
// Method to initialize the TCTI context
void Init_Tcti_Tabrmd_Context();
// Method to finalize the TCTI context
void Finalize_Tcti_Tabrmd_Context();
// Methods to initialize the ESYS context
void Init_Esys_Context();
// Method to finalize the ESYS context
void Finalize_Esys_Context();
// Function to extract hash size
UINT16 GetHashSize(TPMI_ALG_HASH hash);

public:
// Constructor
vTPM(UINT32 id);
// Destructor
~vTPM();
// Setter for nvHandle
void setNvHandle(ESYS_TR handle);
// Getter for nvHandle Pointer
ESYS_TR* getNvHandle();
// Getter for the identifier
UINT32 getIdentifier();
// Function to load an external key
ESYS_TR Load_Public_Key(TPM2B_PUBLIC publicData);
// Function for the creation of the primary key associated with
   th vTPM - used for session authentication
void Create_PK(Key* key);
};
```

As previously mentioned, having a secondary role the functions implemented within the vTPM class have, for the most part, a management function. This fact is due to the fact that the pTPM performs all critical operations, the vTPM has an almost interface function for the Verifier. For this reason, in order not to add too much complexity that would have resulted only for a cosmetic factor, it was preferred to implement the essential functions for the Proof of Concept. The fundamental part of the vTPM class are the variables it has inside. In particular, the `nonceCaller` which is the nonce bound to the Integrity Request and the `identifier`, the unique number that is assigned during the deployment of the vTPM instance. It identifies the vTPM and as a consequence, all its virtual PCR. This is one of the research-key for the Multimap structure which contains the memory location of all the vPCR deployed inside the NVRAM.

Remote Verifier

The Verifier is the remote party that starts the Remote Attestation process. It aims to decide if the Prover can be considered Trusted or Untrusted. It bases the decision on the Integrity Measurements taken during the life-time of the Prover with the so-called Report Evaluation. For this reason in the thesis project, the functions that have been developed aiming to support the new implementation, proving the ability also with the new design of being able to complete the Deep Attestation process successfully.

The actual implementation of this component inside the simulation design is straightforward. As previously mentioned in its functionalities, only two main functions have been implemented: `CreateIntegrityRequest` and `VerifyIntegrityReport`.

```
class Verifier
{
public:
[...]
//Function for the creation of the Integrity Request
IntegrityRequest CreateIntegrityRequest();
// Function for Integrity Report Evaluation
void VerifyIntegrityReport(IntegrityReport integrityReport, Key
   rootKey, Key aik);
[...]
}
```

The first implemented function, `CreateIntegrityRequest`, is the actual function that supports the collection of information. It is the first step to trigger the Prover, composed by the pTPM and vTPM, to collect the information that was previously taken. As can be seen in the main program, first of all, the Integrity Request is created through the function:

```
[...]
cout<<"\n***␣Creating␣the␣Integrity␣Request␣***"<<endl;

IntegrityRequest integrityRequest;
integrityRequest = verifier.CreateIntegrityRequest();

cout<<"\n***␣Sending␣the␣Integrity␣Request␣to␣the␣Agent␣
   ***"<<endl;

[...]
```

The structure of the IntegrityRequest has been previously reported, and as can be seen inside that it is created a nonce to prevent the request to be reused in replay attacks. Apart from the nonce, the other parameters to be sent to the Prover is the list of PCR that the Verifier is interested in. Once the Integrity Request is prepared, it is sent to the Verifier that needs to process it. The pTPM and vTPM work together to collect all the Integrity Measurements taken during the life-cycle of the VM and the VMM until the request is received.

At this point, the Verifier is receiving the IntegrityReport which has to analyze:

```
[...]
cout<<"\n***␣Processing␣the␣Integrity␣Response␣received␣from␣the␣
   Agent␣***"<<endl;

#ifdef DEBUG
getchar();
#endif

verifier.VerifyIntegrityReport(integrityReport,
   pTPM.GetRootKey(), pTPM.GetAIK());

[...]
```

The developed function for the previously mentioned process is the VerifyIntegrityReport. It has three parameters as input. The Integrity Report. It, in particular, contains the structure containing the aggregation of the PCRs values and a signature realized over it. For the validation of the digital signature the Verifier needs the public attestation identity key, which identifies the vTPM and naturally the root public key to validate the key chains, this key instead identify the pTPM involved in the process. For the simulation process, the Verifier can retrieve that information through two methods implemented in the Prover classes.

Apart from those two fundamental classes, the remaining implemented functions are for the management of the class and the context environment.

```
[...]
private:
// Method to initialize the TCTI context
void Init_Tcti_Tabrmd_Context();
// Method to finalize the TCTI context
void Finalize_Tcti_Tabrmd_Context();
// Methods to initialize the ESYS context
void Init_Esys_Context();
// Method to finalize the ESYS context
void Finalize_Esys_Context();
int ErrorHandling(TSS2_RC rc);
[...]
```

5.2.2 Workflow

The `pTPM` class has inside all the critical operations that have been introduced to support the binding between the physical and virtual TPM. First of all the three main characters are defined: **vTPM**, **pTPM** and the **Verifier**. The ACA is involved in general only for the exchange of the certificates and since the thesis is incentrated at a lower level of view the the Prover represented by the virtual and physical TPM will provide the public key directly to the Verifier during the integrity evaluation.

```
*** vTPM - pTPM binding PoC ***
Initialization of the TCTI context successfull
Initialization of the ESYS context successfull
pTPM instantiation successfull

Initialization of the TCTI context successfull
Initialization of the ESYS context successfull
vTPM instantiation successfull

*** Definition of the verifier ***
Initialization of the TCTI context successfull
Initialization of the ESYS context successfull
```

TPM Command Transmission Interface (TCTI) is described in the TPM specifications, and it provides a standard interface for the interaction with TPM through standard commands. Together wIth the TCTI initialization also ESYS context has been initialized since the whole project is based on the utilization of ESAPI.Regarding the initialization, in addition to the definition of the two contexts, only few parameters are defined. In particular for the vTPM the unique *identifier* is assigned. On the other hand in the pTPM is defined the first free memory index inside the NVRAM that will store all the PCRs for the vTPM.

```
pTPM::pTPM(int id)
{
        ...
        // Initialization of the memory index
        freeIndex = TPM2_NV_INDEX_FIRST;
        ...
}
```

Now that all the initializations are completed it is possible to proceed with the Remote Attestation lifecycle following the Deep Attestation process previously described. The first step is the Resource Allocation.

Resource Allocation

During the resource allocation phase, the resources that are represented by the register banks, are defined following the previously described algorithm. For the simulation purpose during this phase, also the registers that will store the VMM measurements are allocated.

```
*** Definition of the NV space ***
Extend NV Index defined: 4293479
Extend NV Index defined: 4293480
Extend NV Index defined: 4293481
...
Extend NV Index defined: 4293488
Extend NV Index defined: 4293489
Extend NV Index defined: 4293490
```

During the resource allocation for each new *nv-index* defined, the results is inserted inside the multimap structure used for this purpose [5.1.1].

The Resource Allocation is not limited to the nvPCRs definition, but in order to complete the Remote Attestation process, the structure that will contain all the useful information needs to be signed by an AIK. For this reason, it is created first of all the primary key from which it will be derived the AIK that will be used during the signature phase.

```
*** Creation of a primary Key for the pTPM ***
Creation of the Key successfull
```

```
*** Creation of the AIK from the primary key ***
Creation of the derived key successfull
```

For the simulation the AIK types has been chose as RSA key with a size of 2048 bits. During the public definition of the key it is chosen also the signature shema, which in this case is RSA-PSS[31] with SHA256 as hash function. At this point all the resources are ready to be utilized during the other phases.

Integrity Measurements

In order to simulate the booting process of the VM, a file has been filled with sample logs of the major phases. Reading the file and extending the correct registers, the process described above has been recreated [Subsection 4.2.3].

```
*** Booting the VM ***
0000 0000 0800 0000 298d f125 b260 ef64
0000 0000 0800 0000 298d f125 b260 ef64
Extension of the space 4293485 successfull
201b df08 15c0 0387 3eed d50e 2700 0000
201b df08 15c0 0387 3eed d50e 2700 0000
Extension of the space 4293485 successfull
...

3135 632d 4145 3835 2d33 3832 3930 4142
3135 632d 4145 3835 2d33 3832 3930 4142
Extension of the space 4293485 successfull
```

Naturally, once the Integrity Response is also created, the VMM measurements will be needed. For this reason in the same way, also some measurements regarding the VMM have been taken. For simulation choice, in particular, the first three PCRs of the SHA1 bank have been designated to store the measurements regarding the *binaries*, *firmware* and *bootloader*.

PCR Number	**Allocation**
0	Binaries
1	Firmware
2	Bootloader

Integrity Request

The VM is now allocated, and it is running, the *Verifier* can now start the Remote Attestation process requesting the **Integrity Report** to the *Prover*. The **Integrity Request** in the simulation is containing a *nonce* to prevent the reply attack and the *pcr-selection* attribute, which contains the bank and register number in which the Verifier is interested in.

```
*** Creating the Integrity Request ***
Nonce = 0123456789
PCR selected:
PCR number #0
PCR number #1
PCR number #2
```

The Integrity Request is sent to the Prover that will process it for the creation of the Integrity Report.

Integrity Report

This is the core part of the whole process. Once the Prover receives the Integrity Request, it extracts the useful parameters, in particular, the nonce and the PCR selection attributes. Both of them will be used in the creation of the Integrity Report. As it was previously described in order to perform a full Deep Attestation, both the VM and VMM will be attested during this phase. For this reason, the quote function will be invoked twice, one to perform the quote operation over the PCR selected by the Verifier for the VM and a second time to perform the quote operation over the PCR that are storing VMM measurements.

```
*** Sending the Integrity Request to the Agent ***
Extracting Integrity Request parameters
Permorming quote operation
Extrapolation of the vPCR
pcrConcatenation
93 38 CC 8 40 65 69 B E2 FC 3A 6A 62 E5 85 FD B1 63 5B 62 F5 68
   D4 B0 8D 9F FF 29 80 4F C3 3B F2 C3 89 75 43 D7 65 CD 45 2 EF
   8 24 A3 92 A0 AF 66 A1 EC 71 93 2B 34 FA FD 9B E9
Digest creation
42 83 B3 89 47 22 E5 1C 15 A8 95 E4 2E 78 93 45 63 F5 81 AA 0D
   5A 1E F4 E7 48 8B 9C 8E 42 C1 5
```

More precisely in the first part as can be seen the PCRs after the extrapolation of their number inside the structure received are read and concatenated. After the PCR concatenation it is calculated the digest of this value. The results will be included inside the structure that will be signed by the TPM. Following the algorithm in the previous section [Sec.5.1.3], the Attested Structure is created.

```
Magic Number
FF544347

Adding Type
FF5443478018

Adding QN signing key
0B1886111C30D8538864F21CDC45436B9DC5787C3F8C2C699AD386BD41DFE98B
```

```
Adding Extra Data
FF54434780180220B1886111C30D8538864F21CDC45436B9DC5787C3F8C2C699A
D386BD41DFE98B0A0123456789

Adding Clock Info
FF54434780180220B1886111C30D8538864F21CDC45436B9DC5787C3F8C2C699A
D386BD41DFE98B0A0123456789000002A8530000000001

Adding Firmware Version
FF54434780180220B1886111C30D8538864F21CDC45436B9DC5787C3F8C2C699A
D386BD41DFE98B0A0123456789000002A8530000000001001B001C

Adding TPMU_ATTEST
FF54434780180220B1886111C30D8538864F21CDC45436B9DC5787C3F8C2C699A
D386BD41DFE98B0A0123456789000002A8530000000001001B001C00030431000
432000434000204283B3894722E51C15A895E42E78934563F581AA0D5A1EF4E74
88B9C8E42C15
```

The TPMU_ATTEST inside the structure includes also the previously created PCR digest. Now it is possible to calculate the hash of the **Attest Structure** to make the digital signature to be sent to the Verifier. All the Hash created are protected by validation tickets that prevent the TPM to accept and sign digest created outside the TPM. For the signature creation is first of all loaded the corresponding AIK.

```
Signature Created
1 D2 47 83 5F E0 7B D4 24 4B 33 BC 73 15 F2 21 35 F0 46 98 BF 0
   BC 37 DA 8E AD 1C 74 E9 7A D7 D4 D3 E2 92 17 DF 89 C1 2F 63
   3E 10 67 DD 3D 8B 84 5E 4A 2A 82 30 CD 12 61 78 ED CD 34 0 9B
   44 EF 20 4C F0 BF B D4 AA E5 4B 1A 9D 27 C8 FC 4E 15 C6 26 6D
   71 2C 2F 61 A 37 C0 C1 A7 27 8F C9 3D 11 17 96 FF 8 94 D9 D 1
   BC C2 5D FF 80 C2 16 78 E4 39 76 4B 13 CB CC 1C 61 36 40 A0
   59 B4 B8 3E 49 94 66 92 28 29 7A 65 53 B4 58 8E 62 8F A8 C6
   3F 98 1E 81 3F DD BE B6 5E D7 9E 4D 56 57 99 7A 26 95 3D 0 B0
   14 51 90 F4 6F B1 BC FE 5D 61 DD FD 4A 22 BF AA 8C 28 B8 EC 0
   24 3B FD BF 20 3D 7 55 5E 18 2F 3B E1 13 D4 97 0 69 6A A7 5D
   81 B 6F 1C 3F C8 23 A2 2C E0 30 39 9D 59 35 79 77 E9 7B 58 9F
   7F 94 F C 28 C6 42 DA F9 BC C1 61 F8 D3 69 66 46 FD 17 6B 38
   E7 C4 DC 3F 4
```

The same procedure (PCR extrapolation, reading the PCRs, creating the Attest Structure and Signature) is repeated for the VMM which will be attested together with the VM.

```
Extrapolation of the vPCR
Size of the index to be defined: 60
pcrConcatenation
4F 80 13 41 E5 70 7C 4A 8A 66 A5 90 15 CA D 58 28 65 5F 37 21 DE
   E5 3E 97 4F 81 C9 58 EE 72 D4 C5 CC C4 50 ED 43 AE FD 4C 8F
   55 70 9A 93 10 8 BE F 51 83 7D 3D CD A 4D DC A9 8A
```

```
Digest creation
DF8FAFCAF12747A4CC97DB17B91837F16293C734E686FBCA4D795197963DB3
Magic Number
FF544347

Adding Type
FF5443478018

Adding QN signing key
0B1886111C30D8538864F21CDC45436B9DC5787C3F8C2C699AD386BD41DFE98B

Adding Extra Data
FF54434780180220B1886111C30D8538864F21CDC45436B9DC5787C3F8C2C699A
D386BD41DFE98B0A0123456789

Adding Clock Info
FF54434780180220B1886111C30D8538864F21CDC45436B9DC5787C3F8C2C699A
D386BD41DFE98B0A0123456789000002A8FD8000000001

Adding Firmware Version
FF54434780180220B1886111C30D8538864F21CDC45436B9DC5787C3F8C2C699A
D386BD41DFE98B0A0123456789000002A8FD800000001001B001C

Adding TPMU_ATTEST
FF54434780180220B1886111C30D8538864F21CDC45436B9DC5787C3F8C2C699A
D386BD41DFE98B0A0123456789000002A8FD800000001001B001C00030431000
43200043400020DF8FAFCAF12747A4CC97DB17B91837F16293C734E686FBCA4D7
95197963DB3

Hashed structure
1CFCD412CE36661887F03C4E8BE10FC1F8D68A2437CE7321DB6490C119E6BA
Signature Created
49 22 DC EC 10 9A 10 3A BE F7 9F BC D1 CB 2 EA F3 B7 F BD ED 4D
    CA 42 B9 24 FD 46 44 99 D5 53 AA 69 F9 2D 65 F 39 BE 97 BF 50
    47 4C 27 D6 6D 2E BF 6A E1 4D CA CF 4 4F 0 12 86 58 D3 74 48
    C6 F2 9C B6 A6 BA 47 B0 8F 3F 84 BD D9 EC 57 CF B9 3 9 A8 4C
    E B0 8D F1 E8 C1 24 5F F0 9E D0 0 F5 7D 92 D8 E2 5 39 41 49
    D4 9C 71 DF D0 57 C3 8 C0 7C DB B 54 2F 45 25 6 6D D5 B0 12
    C1 CA 8B 4E 42 97 DF 65 C9 A1 F4 2 64 A2 AB 74 80 37 E3 8C DE
    36 2D C7 1A 6C 8A 40 81 F1 C1 BB B3 6E 31 73 3 D5 40 EB 30 83
    77 CC C6 27 8 1B DA 15 6B BC 32 DF 48 BE 49 CB AC 1F 25 30 9E
    1F DC B1 9D 0 96 92 67 99 D2 4B AF B0 1 FD 23 98 D1 C0 2A B5
    FA 5 2F 74 68 24 D1 B9 E6 84 6F 2A 90 38 84 F0 F 7B 5F FA 8A
    3D BA BF 18 97 33 9A AB B7 EC C 86 FB F5 87 EA AD E9 91 19 33
    8A 3C B7
```

All those information are now included inside the **Integrity Report** which is sent to the Verifier for the **Report Evaluation**

```
IntegrityReport pTPM::IntegrityRequestFrom_vTPM(UINT32 vID,
   IntegrityRequest integrityRequest)
{
        ...
       integrityReport.attestStructure_VM = attestStructure;
       integrityReport.signatureStructure_VM =
          signatureStructure;
       integrityReport.pcrValues_VM = pcrValues;
        ...
       integrityReport.attestStructure_VMM = attestStructure_VMM;
       integrityReport.signatureStructure_VMM =
          signatureStructure_VMM;
       integrityReport.pcrValues_VMM = pcrValues_VMM;
}
```

Integrity Evaluation

The **Verifier** receives the **Integrity Report** containing all the information it has requested with the **Integrity Request**. With all those information the Prover needs to provide also all the logs collected during **Platform Integrity Measurements**. In this way the Verifier can evaluate if the provided logs are reflecting the PCR values received and if they are an expected value. If all these hyphothesis are respected the VM can be marked as TRUSTED.

```
*** Processing the Integrity Response received from the Agent ***
Verifer loaded public key
*** Veryifing VM Remote Atterstation ***
Received Attest structure
FF54434780180220B1886111C30D8538864F21CDC45436B9DC5787C3F8C2C699A
D386BD41DFE98B0A0123456789000002A853000000000001001B001C00030431000
43200043400020428 3B3894722E51C15A895E42E78934563F581AA0D5A1EF4E74
88B9C8E42C15

Signature received
1 D2 47 83 5F E0 7B D4 24 4B 33 BC 73 15 F2 21 35 F0 46 98 BF 0
   BC 37 DA 8E AD 1C 74 E9 7A D7 D4 D3 E2 92 17 DF 89 C1 2F 63
   3E 10 67 DD 3D 8B 84 5E 4A 2A 82 30 CD 12 61 78 ED CD 34 0 9B
   44 EF 20 4C F0 BF B D4 AA E5 4B 1A 9D 27 C8 FC 4E 15 C6 26 6D
   71 2C 2F 61 A 37 C0 C1 A7 27 8F C9 3D 11 17 96 FF 8 94 D9 D 1
   BC C2 5D FF 80 C2 16 78 E4 39 76 4B 13 CB CC 1C 61 36 40 A0
   59 B4 B8 3E 49 94 66 92 28 29 7A 65 53 B4 58 8E 62 8F A8 C6
   3F 98 1E 81 3F DD BE B6 5E D7 9E 4D 56 57 99 7A 26 95 3D 0 B0
   14 51 90 F4 6F B1 BC FE 5D 61 DD FD 4A 22 BF AA 8C 28 B8 EC 0
   24 3B FD BF 20 3D 7 55 5E 18 2F 3B E1 13 D4 97 0 69 6A A7 5D
   81 B 6F 1C 3F C8 23 A2 2C E0 30 39 9D 59 35 79 77 E9 7B 58 9F
   7F 94 F C 28 C6 42 DA F9 BC C1 61 F8 D3 69 66 46 FD 17 6B 38
   E7 C4 DC 3F 4
Signature Verified
```

```
PCR values:
PCR#0: 93 38 CC 8 40 65 69 B E2 FC 3A 6A 62 E5 85 FD B1 63 5B 62
PCR#1: F5 68 D4 B0 8D 9F FF 29 80 4F C3 3B F2 C3 89 75 43 D7 65
   CD
PCR#2: 45 2 EF 8 24 A3 92 A0 AF 66 A1 EC 71 93 2B 34 FA FD 9B E9
```

Before proceeding with the evaluation of the logs the Verifier verify the signature received using the Public Key of the AIK utilized for the signature. If the result is positive it can proceed to the log evaluation. The Verifier can also extrapolate useful information from the Attest Structure: *nonce*, to verify that is matching the one used inside the **Integrity Request** and the digest of the PCR concatenation, since the PCR values are provided outside the Attest Structure. Once it has evaluated also the logs received it can compare it with the results that is stored inside the PCRs.

```
Checking the logs
PCR #0 93 38 CC 8 40 65 69 B E2 FC 3A 6A 62 E5 85 FD B1 63 5B 62
PCR #1 F5 68 D4 B0 8D 9F FF 29 80 4F C3 3B F2 C3 89 75 43 D7 65
   CD
PCR #2 2 EF 8 24 A3 92 A0 AF 66 A1 EC 71 93 2B 34 FA FD 9B E9

PCR #0 4F 80 13 41 E5 70 7C 4A 8A 66 A5 90 15 CA D 58 28 65 5F
   37
PCR #1 21 DE E5 3E 97 4F 81 C9 58 EE 72 D4 C5 CC C4 50 ED 43 AE
   FD
PCR #2 8F 55 70 9A 93 10 8 BE F 51 83 7D 3D CD A 4D DC A9 8A
```

The Verifier is evaluating, naturally, also the measurements received for the VMM. As can be seen from the output the hash calculated by the Verifier analyzing the logs received from the Prover are matching the one inside the Attest Structure and they are matching the known fingerprints. At this point the Verifier can mark the VM as TRUSTED.
On the other hand if the VM has been tampered and some malicious code is running inside the VM the reported measurements will not match the expected measurements. Even if the attacker is able to provide the logs that would produce the expected value, the PCR will have a different value meaning that the measurements received in the logs are not matching the one taken by TPM.

```
...
Signature Verified

PCR values Received:
PCR#0: 93 38 CC 8 40 65 69 B E2 FC 3A 6A 62 E5 85 FD B1 63 5B 62
PCR#1: 63 05 9A 94 EA 03 FC DE CE C3 DF 61 1E 10 E2 76 A1 12 0B
   CB
PCR#2: 45 2 EF 8 24 A3 92 A0 AF 66 A1 EC 71 93 2B 34 FA FD 9B E9



Checking the logs:
```

```
Expected
PCR #0 93 38 CC 8 40 65 69 B E2 FC 3A 6A 62 E5 85 FD B1 63 5B 62
PCR #1 F5 68 D4 B0 8D 9F FF 29 80 4F C3 3B F2 C3 89 75 43 D7 65
   CD
PCR #2 2 EF 8 24 A3 92 A0 AF 66 A1 EC 71 93 2B 34 FA FD 9B E9

PCR #0 4F 80 13 41 E5 70 7C 4A 8A 66 A5 90 15 CA D 58 28 65 5F
   37
PCR #1 21 DE E5 3E 97 4F 81 C9 58 EE 72 D4 C5 CC C4 50 ED 43 AE
   FD
PCR #2 8F 55 70 9A 93 10 8 BE F 51 83 7D 3D CD A 4D DC A9 8A
```

In the above example firmware measures have been changed and as can be seen they are not matching the expected values or the ones calculated by the Verifier

Chapter 6

Conclusions and Future Work

In this chapter are presented the results and conclusions obtained, pointing out the point of strength and weaknesses of the implemented solution and how it can be improved in future implementations. In particular, in section 6.1 it will be presented the conclusion and some comments regarding the obtained results, while in section 6.2 it will be given some possible future developments.

6.1 Conclusions

The thesis project was introduced to be part of a significant area of research: Cloud Computing. Cloud Computing has fundamentally changed the way how resources are provided and can be accessed. It is a compelling paradigm, offering many resources at low cost including processing, storage and data management. However, it has changed the attack surface to which it is exposed, bringing to light new security and privacy issues to be addressed. For this reason, many solutions have been proposed and discussed by different realities such as NIST and CSA. What it was mainly proposed were solutions based primarily on software implementations. Software implementations that are going to provide a means to trust the execution environment over which all the applications are based on.

Trust is the critical element to have correct and expected behaviour from the cloud environment. Trust that is not only important in a virtual environment, but that is relevant in any machine on which we are making software solutions work. For this reason, the TCG has developed a precise and detailed methodology to be able to obtain it. All this led to the development of the TPM, the means to obtain the desired result. However, the change of scenario has led most of the services to be hosted in virtual machines or more generally in cloud environments. Since it is not possible to have an equal relationship between physical and virtual resources, which is in fact clearly in contrast with what is the paradigm of cloud computing, the TCG had to change its specifications in order to be also adopted in these environments.

The solutions that were proposed, however, were, as already mentioned, mainly software, and were slightly unrelated to what was the true potential of the TPM and its physical implementation. For this reason, we began to think of a solution to extend the protection provided by a trusted device to a virtualize environment. The final goal is to create a strong binding between one or more instances of vTPM and a single pTPM. In this way, end-users can benefit from hardware bound vTPM identity, and the cloud platform can exploit the Deep Attestation process. The thesis project is part of a more significant project, for this reason, the aim of this work was the investigation of PCR bindings for Deep Attestation use. PCRs are one of the essential elements in the attestation process since they store the aggregate measurements taken during the life cycle of the virtual object and their value reflect the actual state of the machine. During the investigation, more than a single solution was implemented. Different internal controls and operations have been considered and developed. At the end of the development of the different solutions, several factors were taken into consideration, including overhead created, scalability and security of the implementation.

The proposed solution was considered the best as it provides a classic Quote operation which is totally transparent to those who invoke it. Although it is necessary to interact with a virtual instance of the TPM, the critical operations that concern the updating of the PCR registers are managed by the physical TPM, which substantially increases the general security of the implementation, which is the primary objective of the thesis. What it has been obtained is, therefore, a binding between vTPM and pTPM that allows the virtual instances of the TPM to be able to instantiate and save the measurements made for the remote attestation directly on a physical resource. Furthermore, this binding also provides to be able to access these resources securely and to share the results obtained with the external Verifier without them being able to be modified in any way. The same result is therefore obtained as if the virtual machine had its own physical TPM and could directly use its physical PCRs.

In addition to the solution obtained, two other implementations were mainly considered. The TPM specifications are very specific and in fact, do not leave much space for free implementations for obvious reasons. However, these two implementations had two significant drawbacks. The first introduced an unnecessary overhead and a meaningful complexity which made its use complicated and expensive in terms of resources. The second solution even if with a minor overhead, did not provide for a classic interaction with vTPM and needed secondary functions to be totally transparent to the Verifier. All this, therefore, introduced complexity and interrupted the chain of trust that had been created between the pTPM and the vTPM. However, the proposed solution is not without drawbacks. This is due to the fact that at the base of the implementation, there are two strong hypotheses in firm contrast to each other. On the one hand, we have the cloud computing paradigm, which involves the sharing of physical resources. On the other hand, we have the remote certification which is based among the many factors involved on a digital signature placed above the value of the PCR to declare that it belongs to a specific TPM. The implementation bottleneck resides in the sharing a physical resource for critical operations which leads to delays when the primary resource needs to be involved in the operations. The second drawback of the solution concerns physical limitations instead. The physical TPM has a limited NVRAM which drives to a limited number of vTPMs that it can manage.

6.2 Future Work

During the implementation study, various options were considered and strengths and weaknesses considered. By analyzing the solution's weaknesses, some improvements can certainly be made to strengthen and make the solution found scalable. The major drawback of the solution is the limited space of the NVRAM memory. It mainly depends on the TPM chosen and can be the real bottleneck by significantly limiting the number of vTPMs that can be managed by a single physical TPM. However, the specifications of the TCG allow under precise specifications to also be able to use external memory to replace the NVRAM. This choice naturally introduces some security issues. The data that is saved on the PCRs that are now implemented on the NVRAM memory inside the TPM could be accessible externally if the necessary precautions were not taken.

Use of an external memory would probably involve the use of an encryption based on a symmetric key in order to be able to save the measurements outside the TPM itself safely. The choice would fall on a symmetric key being faster in terms of data encryption and decryption rather than an asymmetric key. Within the Deep Attestation routine, however, it should be taken into consideration that the data was once directly accessible to the TPM, now instead it must be loaded from an outside resource, decrypted and only at that point it is accessible in order to carry out the required operations. All these additional steps would add slight routine delays which could make vTPM simultaneous access to the TPM problematic. It would therefore be necessary to analyze different types of memory types, their access speed and decide on a symmetric key length that can guarantee a certain level of security and at the same time not create delays that can make the simultaneous access of the physical resource unusable from multiple virtual instances.

Another critical aspect of using a physical TPM in a cloud environment is the fact that virtual machines within data centres or in general are often subject to migration. This fact is a critical point since in general, this would not be possible due to the nature of PCRs that cannot be programmed but can only be updated through the Extend operation. This time, however, the choice to implement PCRs in NVRAM memory is in support of the solution. If in fact by nature the PCRs are not programmable, the PCRs inside the NVRAM, even if they reproduce the functioning of the PCR in its entirety, can be initialized. This fact, therefore, makes the migration possible without adding complexity to the operation or completely invalidating the aggregation of measures within it. It would, therefore, be interesting to add in the routine that manages the migration of the instance of a virtual TPM, which includes key migrations and much more, also a function that manages the loading of PCR coming from another physical TPM. A routine complementary to the one already implemented which allocates for the first time the resources necessary for a virtual machine.

The last change that could be made to the project is to integrate the new commands developed in a parallel project for key management to the Quote function that was implemented instead in this thesis. However, this modification relates to the final PoC of the overall project rather than to the more specific thesis project concerning the binding between virtual PCRs and their instances in the physical TPM. Nevertheless, it must be considered because the Remote Attestation is also based on the use of an asymmetric key used for the digital signature of the Integrity Report provided by the TPM.

Appendix A

Installation Guide

The thesis project is based on the *Linux TPM2 & TSS2 Software* github project, which gives you all the repository and simulators for the implementation of a virtual TPM and all the necessary tools and libraries.

In order to replicate a fully virtual implementation of the environment that TPM needs, it is necessary the following infrastructure:

- **tpm2-tools**. The source repository for the TPM (Trusted Platform Module) 2 tools.
- **tpm2-tss**. OSS implementation of the TCG TPM2 Software Stack (TSS2).
- **tpm2-abrmd**. TPM2 Access Broker & Resource Management Daemon implementing the TCG spec.
- **tpm2-tcti-uefi**. TCTI module for use with TSS2 libraries in UEFI environment.
- **tpm2-tss-engine**. OpenSSL Engine for TPM2 device

Since it is a project in continuos evolution, some of the sub-projects depend on each other. To keep track of the dependencies is useful to check the dependency matrix in continues update on the following link: `https://github.com/tpm2-software/tpm2-tools/wiki/Dependency-Matrix`.

A.1 tpm2-tss

This is the most important repository since it implements TPM2 Software Stack (TSS) described previously implemented by the Trusted Computing Group (TCG). Also for this installation some packages are required, some of them should be yet installed once other repositories have been installed.

- GNU Autoconf

- GNU Autoconf Archive, version ¿= 2017.03.21
- GNU Automake
- GNU Libtool
- C compiler
- C library development libraries and header files
- pkg-config
- doxygen
- OpenSSL development libraries and header files, or optionally libgcrypt
- libcurl development libraries

For the testing purpose of the project we will enable only ESAPI and for this reason the solution can work with either openSSL or libgcrypt (please include also libgcrypt-dev for some dependencies, for any error related to AM_PATH_LIBGCRYPT macro use the following link as a reference: https://github.com/tpm2-software/tpm2-tss/issues/1365)

To build the solution, once installed all the dependencies, the following code can be used:

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

A.2 tpm2-abrmd

This repository implements the daemon for the TPM2 Access Broker & Resource Manager from the TCG. The daemon should be started with the OS boot process. The communication between the daemon and the clients supporting the TPM is done by the combinational usage of the DBus and Unix Pipes. However in the project there is a built-in library to simplify the configuration: **libtcti-tabrmd**. The initialization function is hard-coded to connect the tabrmd on the system bus. This is the most simple configuration and most common configuration. For any specific clarification can be consulted the manual page inside the repository at the following points: TSS2-TCTI-TABRMD(7) and TSS2_TCTI_TABRMD_INIT(3).

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**. As common security practice the daemon can be run as unpriviliged user, which requires creating a user account and group. The account and associated group must be created before running the daemon as follow:

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

In order to build the solution from the repository run the following code for the configuration and bootstrap:

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

In order to run the daemon use the following command:

```
$ sudo -u tss /usr/local/sbin/tpm2-abrmd --tcti=mssim
```

Once installed and running we can test the correctness of the dbus daemon. The purpose of the following command is to list all names on the system bus. In order to keep the output manageable we can filter it by the expected value:

```
dbus-send --system --dest=org.freedesktop.DBus
   --type=method_call --print-reply /org/freedesktop/DBus
   org.freedesktop.DBus.ListNames
```

From the above command the expected output is similar to the following:

```
method return time=1592409144.945722 sender=org.freedesktop.DBus
   -> destination=:1.73 serial=3 reply_serial=2
array [
...
string "com.intel.tss2.Tabrmd"
...
]
```

This means that the daemon claimed correctly the system bus. However we can perform one more sanity check in case the bus seems unresposive or in a hung state:

```
dbus-send --system --dest=com.intel.tss2.Tabrmd
   --type=method_call --print-reply /com/intel/tss2/Tabrmd/Tcti
   org.freedesktop.DBus.Introspectable.Introspect
```

The output of the code should be similar to the following:

```
method return time=1592409332.425811 sender=:1.69 ->
   destination=:1.74 serial=7 reply_serial=2
string "<!DOCTYPE␣node␣PUBLIC␣"-//freedesktop//DTD D-BUS Object
   Introspection 1.0//EN"
"http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd">
<!--␣GDBus␣2.56.4␣-->
<node>
<interface␣name="org.freedesktop.DBus.Properties">
<method␣name="Get">
<arg␣type="s"␣name="interface_name"␣direction="in"/>
<arg␣type="s"␣name="property_name"␣direction="in"/>
<arg␣type="v"␣name="value"␣direction="out"/>
</method>
<method␣name="GetAll">
<arg␣type="s"␣name="interface_name"␣direction="in"/>
<arg␣type="a{sv}"␣name="properties"␣direction="out"/>
</method>
<method␣name="Set">
<arg␣type="s"␣name="interface_name"␣direction="in"/>
<arg␣type="s"␣name="property_name"␣direction="in"/>
<arg␣type="v"␣name="value"␣direction="in"/>
</method>
<signal␣name="PropertiesChanged">
<arg␣type="s"␣name="interface_name"/>
<arg␣type="a{sv}"␣name="changed_properties"/>
<arg␣type="as"␣name="invalidated_properties"/>
</signal>
</interface>
<interface␣name="org.freedesktop.DBus.Introspectable">
<method␣name="Introspect">
<arg␣type="s"␣name="xml_data"␣direction="out"/>
</method>
</interface>
<interface␣name="org.freedesktop.DBus.Peer">
<method␣name="Ping"/>
<method␣name="GetMachineId">
<arg␣type="s"␣name="machine_uuid"␣direction="out"/>
</method>
</interface>
<interface␣name="com.intel.tss2.TctiTabrmd">
<method␣name="CreateConnection">
<arg␣type="ah"␣name="fds"␣direction="out"/>
<arg␣type="t"␣name="id"␣direction="out"/>
```

```
</method>
<method␣name="Cancel">
<arg␣type="t"␣name="id"␣direction="in"/>
<arg␣type="u"␣name="return_code"␣direction="out"/>
</method>
<method␣name="SetLocality">
<arg␣type="t"␣name="id"␣direction="in"/>
<arg␣type="y"␣name="locality"␣direction="in"/>
<arg␣type="u"␣name="return_code"␣direction="out"/>
</method>
</interface>
</node>
"
```

A.3 tpm2-tools

The **tpm2-tools** project aims to provide both low-level and aggregate command to have access to the functionalities that the *TPM 2.0* device is giving. As previously mentioned **tpm2-tools** as a strong dependencies on the other project so before installing the tool is worth to check the dependencies to satisfy. This distribution relies on the following dependencies that needs to be installed before its installation.

- GNU Autoconf
- GNU Automake
- GNU Libtool
- pkg-config
- C compiler
- C Library Development Libraries
- ESAPI - TPM2.0 TSS ESAPI library
- OpenSSL libcrypto library
- Curl library

In order to install the components that are not present in the test machine follow your linux distribution system syntax. In addition to the OS dependencies for the installation, the tool require also the following dependencies:

- tss
- pandoc (optional since for manual page generation)
- abrmd (optional in general but not for the purpose of the thesis since)

Installation of the dependencies from the same project, tss and abrmd, are already covered in this guide. Dependenxies based on linux distribution must be checked manually following the linux dependant sintax. To obtain instead the tpm2-tools sources first of all we need to clone the repository inside a folder.

```
$ git clone https://github.com/tpm2-software/tpm2-tools
```

To compile and build the tool execute the following commands:

```
$ cd tpm2-tss
$ ./bootstrap
$ ./configure --prefix=/usr
$ make -j$(nproc)
$ sudo make install
```

At this point we would be able to test the functionalities of our hardware tpm directly, however it is not recommended the testing and implementation of against a real TPM because of the impossibility or highly difficult procedure to clear something lock inadventertly inside the NVRAM.

A.4 tpm2-tcti-uefi

In a UEFI environment the TCTI module is used by TSS. Below the necessary dependencies for the installation:

- GNU Autoconf
- GNU Automake
- C compiler
- linker
- gnu-efi (>= 3.0.8)
- tpm2-tss

Installation of the solution is the following one:

```
$ gitclone https://github.com/tpm2-software/tpm2-tcti-uefi.git
$ cd tpm2-tcti-uefi
$ ./bootstrap
$ ./configure
$ make
$ make install
```

A.5 tpm2-tss-engine

This project is not strictly necessary for the thesis project build but it offers a cryptographic engine for OpenSSL for TPM2.0 using the TSS software stack. The peculiarity is the support for RSA and ECDSA signature. It is useful for analyzing the signature generated during the thesis project or as an additional tool for testing the TPM capabilities.

Dependencies for the build:

- GNU Autoconf
- GNU Autoconf Archive
- GNU Automake
- GNU Libtool
- C compiler
- C library development libraries and header files
- pkg-config
- OpenSSL >= 1.0.2
- tpm2-tss >= 2.2.2
- pandoc

In order to build the solution instead, it is possible to use the following commands:

```
$ gitclone https://github.com/tpm2-software/tpm2-tss-engine.git
$ cd tpm2-tss-engine
$ ./bootstrap
$ ./configure --enable-debug
$ make check
$ make
$ make install
$ sudo ldconfig
```

A.6 Simulator: MS TPM 2.0

For the thesis project has been used the official TCG reference implementation of the TPM 2.0 implementation. The project has also a Visual Studio solution, for the thesis project it will be followed the Linux build solution since the project was developed under a Linux environment. The dependencies for a correct build are the following:

- autoconf-archive
- pkg-config
- libssl-dev

For a linux:

```
$ gitclone https://github.com/microsoft/ms-tpm-20-ref.git
$ cd ms-tpm-20-ref
$ ./bootstrap
$ ./configure --enable-debug
$ make check
$ make
$ make install
$ sudo ldconfig
```

Appendix B

User Manual

Prerequisites

All the PoC is based on the usage of TPM 2.0. In this case it will be used a simulator. The tpm simulator is based on 5 general components:

- **TPM**, which can be a firmware TPM, a discrete TPM or in in this case a TPM simulator.
- **Resource Manager**. It is used for object life-cycles in the TPM. They are generally called abrmd and in linux kernel abrmd in the tpm driver.
- **System APIs**. They can be the lowlevel SAPI or ESAPI.
- **TCTI**, a trasmission interface library
- **Client**, which is responsible of the ESAPI usage, like the tpm2-tools project.

For this reason in order to access the TPM the following is needed:

- Create a TCTI context. It will allow to send and receive data to and from the Resource Manager or the TPM itself.
- Provide a TCTI context once the ESYS context is initialized.
- Use the ESAPI calls to interact with the TPM and send commands to the TPM.

Testing

First, launch the TPM 2.0 simulator through the script *run_simulator.sh* present in the main folder:

```
$ sudo ./run_simulator.sh
```

The expected output from the previous command is the following:

```
Terminate with CTRL + C ...
TPM command server listening on port 2321
Platform server listening on port 2322
Client accepted
Client accepted
```

To compile the example:

```
$ make
```

To run the example, move to /build/ folder and run:

```
$ ./build/run_test
```

Depending if you want to run the solution in interactive mode or not the output will be different. However the expected starting is the following:

```

*** vTPM - pTPM binding PoC ***
```

If the output seems in an hung state and the carriage-return button is not starting the simulation please check if the *run_simulator* script is still running in the other CLI

Interactive mode

Remove the DEBUG variable from the build step in the Makefile to launch the test in non-interactive mode. On the other hand, it is possible to follow step by step all the procedure pressing the carriage-return button everytime that the command line interface is waiting for the input to proceed. Below the interested part of the code of *Makefile* to be edited:

```
...
#VARS=-DDEBUG # interactive mode
VARS=
...
```

Classes and scripts

There are two main scripts: *Makefile* and *run_simulator.sh*. The first can be used to compile the solution created, the second one is used to initialize the simulator, the TCTI context and the TPM2 Access Broker & Resource Manager.

The other classes are used to represent the different agents involved in the PoC that are explained in the previous sections.

All the other installations and agents involved are explained in the different in the previous Appendix. However manual pages of the *tpm2-software* project used to implement the TPM simulator and all the other dependecies can be found also at the following links:

- TPM2 Access Broker & Resource Management Daemon implementation: tpm2-abrmd
- The source repository for the TPM (Trusted Platform Module) 2 tools: tpm2-tools
- OSS implementation of the TCG TPM2 Software Stack (TSS2): tpm2-tss
- OpenSSL Engine for TPM2 devices: tpm2-tss-engine
- TCTI module for use with TSS2 libraries in UEFI environment: tpm2-tcti-uefi

Acronyms

AIK Attestation Identity Key.

AS Attestation Server.

CERT Computer Emergency Response Team.

CRTM Core Root of Trust Measurement.

CSA Cloud Security Alliance.

DA Deep Attestation.

EK Endorsement Key.

EPC Enclave Page Cache.

IaaS Infrastructure as a Service.

IAD Integrated Access Device.

IDM Identity Device Manager.

KMS Key Management Server.

MLE Measured Launch Environment.

NIST National Institute of Standards and Technologies.

OAT Open Attestation Toolkit.

OS Operatying System.

PaaS Platform as a Service.

PCR Platform Control Register.

PRM Processor Reserved Memory.

SaaS Software as a Service.

SRK Storage Root Key.

TBB	Trusted Building Block.
TCG	Trusted Computing Group.
TPM	Trusted Platfomr Module.
TT	Trunsitive Trust.
UI	User Interface.
VM	Virtual Machine.
VMM	Virtual Machine Manager.

Bibliography

[1] W. Arthur and D. Challener, “A Practical Guide to TPM 2.0: using the Trusted Platform Module in the new age of security”, Apress, 2015, ISBN: 9781430265849

[2] TCG, “Trusted Platform Module Library Part 1: Architecture”, 2019. Rev-2.0 https://trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-1-Architecture-01.16.pdf

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

[4] B. Grobauer, T. Walloschek, and E. Stocker, “Understanding Cloud Computing Vulnerabilities”, IEEE Security & privacy, vol. 9, June 2010, pp. 50–57, DOI 10.1109/MSP.2010.115

[5] P. Mell and T. Grance, “The NIST Definition of Cloud Computing. National Institute of Standards and Technology”, NIST special publication, September 2011, pp. 2–3, DOI 10.6028/NIST.SP.800-145

[6] ITU-T SG13, “Y.3500: Information Technology - Cloud Computing - Overview and Vocabulary)”, August 2014, https://www.itu.int/rec/T-REC-Y.3500-201408-I

[7] P. Pandya and R. Rahmo, “Cloud Computing Architecture and Security Concepts”, Cloud Computing Security: Foundations and Challenges (P.Stavroulakis and M.Stamp, eds.), pp. 199–210, CRC Press, 2016

[8] C. S. Alliance, “Top Threats to Cloud Computing: Egregious Eleven”, CSA Official Press Release, June 2019. https://cloudsecurityalliance.org/artifacts/top-threats-to-cloud-computing-egregious-eleven

[9] T. Grance and W. Jansen, “Guidelines on Security and Privacy in Public Cloud Computing”, Special Publication (NIST SP), December 2011, DOI 10.6028/NIST.SP.800-144

[10] A. Gopalakrishnan, “Cloud Computing Identity Management”, SETLabs briefings, vol. 7, May 2009, pp. 45–54

[11] S. M. Kumar and P. Rodrigues, “A Roadmap for the Comparison of Identity Management Solutions Based on State-of-the-Art IdM Taxonomies”, International Conference on Network Security and Applications, Chennai (IN), July 23-25, 2010, pp. 349–358, DOI 10.1007/978-3-642-14478-3_36

[12] M. K. Srinivasan and P. Rodrigues, “Analysis on Identity Management Systems with Extended State-of-the-art IDM Taxonomy Factors”, International Journal of Ad hoc, Sensor & Ubiquitous Computing (IJASUC), vol. 1, December 2010, pp. 62–70, DOI 10.5121/ijasuc.2010.1406

[13] W. A. Jansen, “Cloud Hooks: Security and Privacy Issues in Cloud Computing”, 2011 44th Hawaii International Conference on System Sciences, Kauai,

(HI, USA), 4-7 January, 2011, pp. 1–10, DOI 10.1109/HICSS.2011.103

[14] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing", 2nd international workshop on hardware and architectural support for security and privacy, Tel-Aviv (IL), June, 2013, pp. 1–8, DOI 10.1145/2487726

[15] B. Gassend, G. E. Suh, D. Clarke, M. Van Dijk, and S. Devadas, "Caches and Hash Trees for Efficient Memory Integrity Verification", The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003., Anaheim, (CA, USA), February 8-13, 2003, pp. 295–306, DOI 10.1109/HPCA.2003.1183547

[16] N. F. B. Awang, "Trusted Computing - Opportunities & Risks", 5th International Conference on Collaborative Computing: Networking, Applications and Worksharing, Washington (DC, USA), November 11-14, 2009, pp. 1–5, DOI 10.4108/icst.collaboratecom2009.8402

[17] R. Savino, "Open CIT Product Guide." `https://github.com/opencit/opencit/wiki/Open-CIT-3.2.1-Product-Guide`

[18] "OpenAttestation SDK." `https://wiki.openstack.org/wiki/OpenAttestation`

[19] J. Schiffman, H. Vijayakumar, and T. Jaeger, "Verifying System Integrity by Proxy", International Conference on Trust and Trustworthy Computing, Vienna (AU), June 13-15, 2012, pp. 179–200, DOI 10.1007/978-3-642-30921-2_11

[20] R. Perez, R. Sailer, L. van Doorn, *et al.*, "vTPM: Virtualizing the Trusted Platform Module", 15th Conf. on USENIX Security Symposium, Vancouver (CA), July 31 - August 4, 2006, pp. 305–320. `https://www.usenix.org/legacy/events/sec06/tech/full_papers/berger/berger_html/index.html`

[21] L. Chen, M.-F. Lee, and B. Warinschi, "Security of the Enhanced TCG Privacy-CA Solution", International Symposium on Trustworthy Global Computing, Madrid (ES), August 31 - September 1, 2011, pp. 121–141, DOI 10.1007/978-3-642-30065-3_8

[22] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", IETF, May 2008, DOI 10.17487/RFC5280

[23] I. Union, "ITU-T recommendation x. 509 (08/97) - Information Technology - Open Systems Interconnection - The directory", Authentication framework, August 2001. `https://www.itu.int/rec/T-REC-X.509-199708-S/en`

[24] TCG, "TCG Infrastructure Working Group A CMC Profile for AIK Certificate Enrollment", 2011. Rev-7, `https://trustedcomputinggroup.org/wp-content/uploads/IWG_CMC_Profile_Cert_Enrollment_v1_r7.pdf`

[25] ARM, "Security Technology Building a Secure System Using Trustzone Technology (white paper)", 2009, `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/`

[26] D. D. Clark and D. R. Wilson, "A Comparison of Commercial and Military Computer Security Policies", 1987 IEEE Symposium on Security and Privacy, Oakland, (CA, USA), April 27-29, 1987, pp. 184–184, DOI 10.1109/SP.1987.10001

[27] TCG, "Trusted Platform Module Library, Part 2: Structures", 2019. Rev-2.0, `https://trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.`

0-Part-2-Structures-01.38.pdf

[28] TCG, "TCG TSS 2.0 Enhanced System Level API (ESAPI) Specification", 2019. Rev-5.0, https://trustedcomputinggroup.org/wp-content/uploads/TSS_ESAPI_v1p0_r08_pub.pdf

[29] T. P. C. Platform, "TPM Profile (PTP) Specification". TCG, 2017. Revision 37, https://trustedcomputinggroup.org/wp-content/uploads/PC-Client-Specific-Platform-TPM-Profile-for-TPM-2p0-v1p04_r0p37_pub-1.pdf

[30] V. Haldar, D. Chandra, and M. Franz, "Semantic Remote Attestation: A Virtual Machine Directed Approach to Trusted Computing", USENIX Virtual Machine Research and Technology Symposium, San Jose, (CA,USA), May 6-7, 2004. https://www.usenix.org/legacy/publications/library/proceedings/vm04/tech/haldar/haldar.pdf

[31] J. Jonsson and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1." RFC-5246, February 2003, DOI 10.17487/RFC3447