POLITECNICO DI TORINO

Master's degree course in Computer Engineering

Master's Degree Thesis

Use of Trusted Computing techniques to counteract Cybersecurity attacks in Critical Infrastructures

Supervisors
prof. Antonio Lioy
dr. engr. Diana Gratiela Berbecaru
dr. Ignazio Pedone

Candidate
Enrico Bravi

Academic Year 2021-2022

Alla mia famiglia, che mi ha sempre supportato e incoraggiato

Abstract

Nowadays to manage critical infrastructures there are largely adopted paradigms such as Cloud Computing, Fog Computing, and Edge Computing. They introduce several advantages, like ensuring great flexibility, availability and reducing management costs. These goals are mostly achieved thanks to the advantages of virtualization technologies. Despite these techniques introduce several advantages in terms of performance, they introduce also several security threats like attacks against software integrity. To mitigate these kinds of threats can be used *Trusted Computing* techniques like Remote Attestation (RA) which permits a third party (*Verifier*) to verify the software and configurations' integrity of a platform (*Attester*) to determine its trustworthiness. To perform RA there are several techniques that can be based on some secure hardware (e.g. TPM and TEE based) or based only on software solutions (e.g. Pioneer). The problem addressed by this work is the lack of a generic model for remote attestation which makes it difficult to attest different objects and aggregate different attestation technologies. The proposed solution consists of a new version of an already proposed system called *Trust Monitor*. The purpose of this thesis is to propose a new design and a new implementation of this platform in order to can reach the highest level of flexibility that this system can offer and to be able to integrate this platform into the largest number of possible scenarios. The solution proposed permits to be independent of the objects on which performing RA (physical nodes, virtual machines, containers, pods, enclaves), introducing an object model completely general in order to can save the necessary information for each kind of possible entity. In addition, it permits to reach the independence of the platform from the RA technologies (Keylime, Open Attestation, Intel SGX) used to perform this verification thanks to a dynamic load of ad hoc plugins for each technology used.

Summary

The following thesis is carried out in the context of Cloud Computing, which is a computing paradigm that has had dizzying growth in recent years. This growth is justified by the great advantages that this paradigm introduces like ensuring great flexibility and availability and reducing the management cost making available to the client the possibility to not directly manage the hardware used to deploy the IT infrastructure but only the high-level services. These goals are achieved thanks to the advantages of virtualization technologies which are highly used in a Cloud Computing scenario. These technologies can be of full virtualization (e.g. VMware, Hyper-V, KVM) that permits to create and manage virtual machines, or they can be lightweight virtualization (e.g. Docker) that permits to create and manage containers. These are elements that allow to deploy services as a set of micro-services, which permits reaching a high level of availability and flexibility. Despite the cloud computing paradigm introduces several advantages in terms of performance, it introduces also several security threats such as malicious insider, isolation failure, and economic denial of service, which expose companies and users to important privacy and security dangers. For all these reasons it is very important the possibility of verifying the integrity and the correct configuration of the software running on cloud nodes in order to be able to detect tampering and react accordingly.

The mechanism used for this integrity checking is *Remote Attestation* (RA) which permits a third party to verify if the software running on both physical nodes and virtual nodes, realized as tradition or lightweight virtual machines, is the expected one in order to determine the trustworthiness of the system. In order to perform RA, there are several techniques and technologies which can be based on some secure hardware or based only on software solutions. These solutions can be adopted in different scenarios, from IoT to large cloud infrastructures. One of these scenarios is the case of Network Function Virtualization (NFV) which exploits the advantages of virtualization in order to offer an infrastructure with virtualized components, called Virtualized Network Functions (VNFs), instead of physical ones. In the particular case of NFV, the European Telecommunications Standards Institute (ETSI) has defined several standards regarding the components and how should take place the process of RA of VNFs, defining a particular component, called Trust Manager, to integrate into the NFV infrastructure in order to verify the integrity of VNFs.

The TORSEC research group proposed a system called *Trust Monitor*, which can be considered an implementation of the Trust Manager, in order to be able to verify the integrity of an NFV infrastructure, in particular running Virtual Network Security Functions (vNSFs) which implements specific components with security purposes. The purpose of this thesis is to propose a generic attestation model in order to attest different objects and aggregate different attestation technologies. In order to reach this result, it has been proposed a new design and implementation of the Trust Monitor. This is necessary in order to can reach the highest level of flexibility that this system can offer. The goal of this work was to realize a system that can add an abstraction layer above the RA one in order to make the Trust Monitor independent from the objects that need integrity verification (physical nodes, virtual machines, containers, pods, enclaves), and also independent from the RA technologies used to perform this verification (Keylime, Open Attestation, Intel SGX).

Functional and performance tests have been performed on the proposed solution in order to verify that the system works as expected and that is able to scale depending on the number of objects to manage.

Acknowledgements

I would like to express my gratitude to Prof. Antonio Lioy and Dr. Engr. Diana Gratiela Berbecaru for the opportunity they gave me of working on this thesis.

I would also like to thank Dr. Ignazio Pedone for his insightful comments and suggestions.

I would like to extend my sincere thanks to Dr. Silvia Sisinni for the technical support she offered me.

My most important thanks go to my family to whom I am deeply grateful for always believing in me.

Finally, I would like to thank my dearest friends who accompanied me during my studies, alleviating the resulting tiredness and tension.

Contents

1 Introduction 6

2 Trusted Computing and Remote Attestation 9

2.1 Trusted Computing . . . 9

2.1.1 Trusted Platform Module (TPM) . . . 10

2.2 Roots of Trust (RoTs) . . . 12

2.2.1 Root of Trust for Storage (RTS) . . . 12

2.2.2 Root of Trust for Measurement (RTM) . . . 12

2.2.3 Root of Trust for Reporting (RTR) . . . 12

2.3 Trusted Boot . . . 12

2.4 Remote Attestation Techniques . . . 14

2.4.1 Remote Attestation . . . 14

2.4.2 Harware-based Attestation . . . 15

2.4.3 Software-based Attestation . . . 17

2.4.4 Hybrid Attestation . . . 18

2.5 Trusted Execution Environment (TEE) overview . . . 18

2.5.1 Intel SGX . . . 19

3 Remote attestation standards and use cases 21

3.1 Device Identifier Composition Engine (DICE) . . . 21

3.2 Remote Attestation Procedures (RATS) . . . 23

3.2.1 Topological Pattern . . . 25

3.3 Trusted Execution Environment Provisioning (TEEP) . . . 26

3.3.1 Architecture . . . 26

3.4 ETSI GR NFV-SEC 018 . . . 27

3.4.1 Network Functions Virtualization . . . 27

3.4.2 NFV Remote Attestation Architecture . . . 28

4 Trust Monitor (TM) 31
4.1 Overview and Motivation . . . 31
4.2 Architecture . . . 31
4.2.1 Attestation Process . . . 33
4.3 Trust Monitor 2.0 . . . 34
4.3.1 Keylime Attestation Framework . . . 34
4.3.2 Keylime Attestation Driver . . . 36
4.3.3 Whitelist Web Service . . . 38
4.4 Criticalities and open challenges of the classic version . . . 38

5 Trust Monitor Redesign 39
5.1 Architecture of new Trust Monitor . . . 39
5.2 Level of Abstraction . . . 40
5.2.1 Attestation Adapters . . . 40
5.3 Description of the Componets . . . 41
5.3.1 TM Core Application . . . 41
5.3.2 Connectors . . . 41
5.3.3 Databases . . . 42
5.3.4 Queues . . . 43
5.4 Interfaces and High-Level Workflow . . . 43
5.4.1 TM Operations . . . 44

6 Trust Monitor Implementation 46
6.1 Tools and Libraries . . . 47
6.1.1 Configuration File . . . 47
6.1.2 APIs Manager . . . 48
6.1.3 Databases . . . 49
6.1.4 Queues . . . 50
6.1.5 Databases' connectors . . . 50
6.2 Low-level workflow of the solution . . . 51
6.2.1 Adapters . . . 51
6.2.2 TM Core Application . . . 52
6.2.3 Adapters' connector . . . 53
6.3 APIs and Operations . . . 53

7 Test and Validation 55
7.1 Testbed . . . 55
7.2 Keylime attestation adapter . . . 55
7.3 Functional tests . . . 57
7.4 Performance tests . . . 62

8 Conclusions and future work 65

Bibliography 67

A User's Manual 70

A.1 System deployment 70

A.1.1 Enabling TLS 71

A.1.2 Keylime 71

A.1.3 Adding attestation adapters 71

A.2 Use of databases 73

B Developer's reference guide 75

B.1 Trust Monitor APIs 75

Chapter 1

Introduction

IT infrastructure security is becoming more and more important in a business context, because the services offered manage sensitive data, or execute code, which could be tampered with and becomes dangerous for the infrastructure itself. The Cloud Computing approach permits the management of several aspects of an IT Infrastructure, from hardware to applications deployed. In this scenario a Cloud Provider can make available the infrastructure to the client, at different levels: it can supply hardware (virtual hardware), so that the client can build its infrastructure as it would be on-premise, but not manage the maintenance of physical nodes. This approach is called Infrastructure as a Service (IaaS). The Cloud Provider could make available directly an application or a service, hiding all the infrastructure details. In this case, the approach is called Software as a Service (SaaS).

The key concept of all cloud computing models is the possibility to offer a flexible service, in this way it becomes easier for the client to access it and easier for the cloud provider to offer and manage it. Over the year this paradigm has become more and more used, raising several security issues. In fact, security is one of the biggest barriers to the wider adoption of cloud computing [1].

The concept of *virtualization* has radically changed the network and cloud world, permitting the optimization of the resources utilization and services provided. virtualization introduces some security features hidden in the concept itself, in fact, the concept of *Virtual Machine* (VM) permits to achieve a high level of isolation at the application level, so each VM has a separate environment from the others, having also its own (virtual) hardware. The entity that permits the virtualization is the *Hypervisor*, which can be executed at several levels of privileges on a physical machine, it depends on what is the purpose of the system. In an industrial context, the Hypervisor generally is a very stripped-down OS that runs on bare metal and executes and manages some "special" processes (VMs). This permits to have accurate control of the physical hardware. In a personal context, this solution could be excessive from a management point of view. For this reason, a Hypervisor can be executed at the application level, as a normal process running over an OS. In this case, there are some considerations on the privileges associated with the Hypervisor because in order to use the hardware has to call the underneath OS which has the actual control of the physical device. There are also other solutions to obtain isolated environments, like *Lightweight Virtualization*, which instead of using software developed to virtualize, exploits several OS features and concepts in order to isolate some processes but at the kernel level, without the hardware Virtualization process. In this context, it introduces the concept of *Container* which is the isolated entity from the machine environment. In this way, the isolation can be different based on the necessity of the moment, because can be, for example, at the network level, at the process level, at the user level, and even combine these levels of isolation. Of course, virtualization introduces also some vulnerabilities [2], and for this reason, some techniques to monitor a virtualized environment are needed.

An important example of how can be used virtualization to add flexibility in a network context is *Network Function Virtualization* (NFV), which permits to move all the services provided by a network provider to a higher level, not making available hardware, because of the difficulty of

managing several tenants that can require very different services (e.g. proxies, firewalls, gateways), but making available *Virtualized Network Functions* (VNFs). This paradigm permits the management of network security by providing security functions called *Virtual Network Security Functions* (vNSFs). Thanks to this technology it is possible to offer services for the security of infrastructures following the concept of *Security as a Service* (SECaaS) [3]. These functions allow to protect the entire infrastructure from malicious intentions that can compromise the security of the entire infrastructure. They are also vulnerable to attack though, in fact, they could be subjected to some kind of manipulation that compromise their behavior.

A possible technique that could be adopted to mitigate these manipulation threats, is *Remote Attestation* [4]. This process is based on the possession of some information about the expected behavior of an entity (*Attester*) of the infrastructure, which has to send its current state, which typically consists of a set of measurements of software components running on the system, to another entity (*Verifier*) that has the duty to confront this data with the already possessed to check if the Attester is in a trusted state or not. This technique can rely on some secure hardware installed on the device, for example, the Trusted Computing Group (TCG) has designed the *Trusted Platform Module* (TPM) [5], which permits to build a chain of trust from the hardware up to the software. In not every case is possible to use a TPM though, in fact, especially for IoT devices that have limited resources, because the system cannot be provided with additional secure chips like the TPM. For these scenarios have been developed some solution that permits to achieve an acceptable level of trustworthiness of devices, based only on the software that runs on the system and some secret securely stored in the device's memory. The *European Telecommunications Standards Institute* (ETSI) has designed a standard [6] for Remote Attestation, specific for NFV environments where it explains the procedure to obtain proof of the trustworthiness of services (VNFs). ETSI proposed an entity called *Trust Manager* which has the purpose to verify the integrity VNFs in NFV infrastructure and permits to centralize all these operations.

The TORSEC research group has developed a system called **Trust Monitor** which is a possible implementation of the Trust Manager entity. It was designed for the remote attestation of physical nodes and VNFs, in particular vNSFs in a SECaaS scenario, of an NFV infrastructure introducing the needed centralization of attestation processes. Even though the Trust Monitor was proposed for a specific SECaaS scenario, the goal of this system is the possibility to be integrated into several kinds of infrastructures in order to be able to manage the remote attestation of more generic objects like VMs or containers, so for a generic Cloud Computing scenario. A great possibility that the Trust Monitor offers is using different remote attestation technologies because its architecture introduces a component called *Attestation Driver*. An Attestation Driver is a module that implements the necessary logic to interface with a particular attestation technology. In this way, it is possible to use several technologies deploying several Attestation Drivers for each technology. The proposed architecture of the Trust Monitor originally implements a verifier which confronts the Integrity Report received from the Attester with the reference values of that particular entity and decides if the object is trusted or not. This could be a problem from the flexibility point of view because the Trust Monitor is dependent on the attestation technologies used.

The purpose of this thesis work is to propose a generic model of remote attestation in order to attest different objects and aggregate different attestation technologies. This has been reached by redesigning the entire architecture of the Trust Monitor in order to achieve a system completely independent from the remote attestation technologies used in the infrastructure and from the objects to be verified for integrity. This redesign would permit obtaining a system that can be deployed in several Cloud Computing scenarios without the modification of the core of the system but only by integrating different components. This is achieved by moving the main remote attestation logic on the Attestation Drivers, called *Attestation Adapters* in this new version, in order to obtain the independence wanted. In addition, the logic of the Verifier is not implemented in the Trust Monitor but it is used the verifier implementation of each technology integrated into the system, so the Trust Monitor has only the purpose to aggregate all the results received from these verifiers. This approach permits adding a level of abstraction to the remote attestation process because all the logic that controls a single attestation technology is moved in the relative adapter so the administrator interface became the Trust Monitor which hides all these sub-processes. The solution proposed provides an adapter for the Keylime attestation framework. In this particular

case, it is a modified version of Keylime which permits the remote attestation of nodes and *pods*, developed for remote attestation in a Kubernetes [7] infrastructure.

The tests performed on the system have been both functional and performance. The results show that the system works as expected and that the overhead time added to the Keylime framework is negligible. In addition, for performance tests, has been deployed a simulated environment that has permitted test the system with up to 36 nodes under attestation, and the result is that the proposed solution scale correctly with an increasing number of objects under attestation.

Chapter 2

Trusted Computing and Remote Attestation

2.1 Trusted Computing

Trusted Computing was introduced in the 1990s to deal with the problem of platform trustworthiness, which is defined as the expectation that the behavior will be a particular one, depending on the purpose of the platform.

Being able to understand if a node is in a trusted or untrusted state, is very important in a cloud environment because nodes are exposed to the outside world and they could be compromised. A node could be manipulated in several ways and a possible one is by software alteration. For this reason, it's needed a mechanism that permits ascertaining if a component of the infrastructure is in the expected state, and can securely communicate with other components. This mechanism has to demonstrate that the node is "trusted", otherwise a node that is identified as "untrusted" should be disconnected from the network and restored.

The *Trusted Computing Group* (TCG) is the organization that develops and promotes tools related to trusted computing, and it has defined the concept of *Trusted Platform* (TP). Moreover, the TCG has proposed a possible implementation of the TP, which relies on an additional chip called *Trusted Platform Module* (TPM), which is widely accepted by the industrial world [8]. Several other devices that fit this definition of "trusted platform", or have a sufficient overlap that it must be considered their contribution to the family's lineage [9]:

- **Secure Coprocessor** which is a subcomponent of a computing system, that executes security-sensitive computation. The key concept of a secure coprocessor is not *security* per se, but instead is building a trusted environment, where sensitive applications can be executed in it, increasing protection against possible attackers (Figure 2.1)
- **Cryptographic Accelerators** that permits to perform intensive cryptographic computation. They are considered trusted computing platforms because they began to store sensitive keys, that permit to obtain features like physical security and programmability.
- **Personal Tokens** which is a dedicated hardware a user can use to perform several operations, such as to authenticate themself, cryptographic operations, etc. Those devices require some physical security, that permits preventing a malicious user to learn enough from a token or modify the token state.
- **Dongles** that is typically used to prevent copying of the software. The main idea is the software runs on a general-purpose system, with a dongle attached, and this software interacts with this device, and it cannot run successfully without the dongle's response. This solution is often provided by a software vendor.

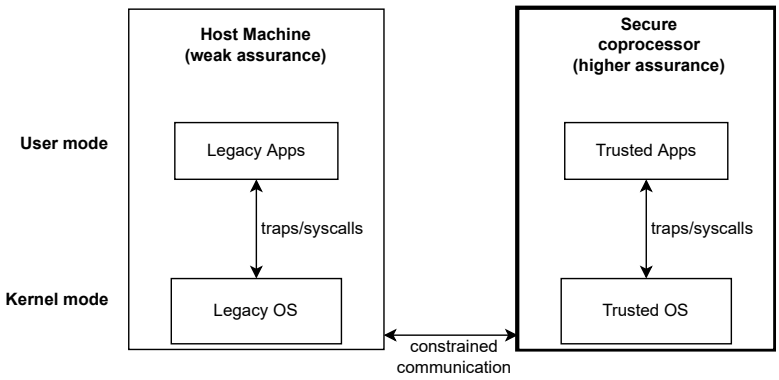

Figure 2.1. Secure Coprocessor (source: [9])

- **Trusted Platform Module** is an independent chip, mounted on the motherboard of a general-purpose machine, that permits to increase in the security of the computation. One advantage of this solution is that it can potentially secure the entire general-purpose machine, on the other hand, providing effective security for an entire system by physically protecting the TPM and leaving the CPU and memory exposed is a delicate matter. This is currently one of the most used solutions at the industrial level.
- **Hardended CPUs** which is the idea to add additional functionality to the CPU instead of adding physical security, in this way it could be possible to transform an entire general-purpose machine into a trusted platform.

2.1.1 Trusted Platform Module (TPM)

Those solutions have been introduced because it become obvious that trying to detect malicious software with software-only solutions, could be easily substituted with some hardware [10]. As reported before, one of the most used hardware solutions is the TPM which was developed to be a minimal chip that could be placed on the PC's motherboard. Another characteristic was to have the instructions set reduced, containing only the necessary functions delegating all the more sophisticated functionalities to the software layer. All those specifications were introduced in the publication of the TPM version 1.1b which contained the following functionalities [11]:

- key generation;
- storage of integrity metrics in special registers named *Platform Configuration Registers* (PCRs);
- reporting of integrity metrics;
- secure authorization;
- cryptographic operations;
- use of *Attestation Identity Keys* (AIKs), related with the TPM identity;

This version of the TPM was introduced by the Trusted Computing Platform Alliance (TCPA), which become the TCG and published the TPM version 1.2 [5] that was aimed to address the following major issues in the industry [11]:

- *Identification of devices*: It was necessary because, before the release of the TPM specifications, devices were identified by MAC addresses and IP addresses, which are not secure identifiers.

- *Secure generation of keys*: For creating secure keys, a hardware random-number generator is needed, because many security solutions have been broken.

- *Secure storage of keys*: TPM permits to keep secure the generated keys, particularly from software attacks.

- *NVRAM storage*: Having a non-volatile RAM permits the TPM to maintain a certificate store, even if a hard drive is rewritten.

- *Device health attestation*: TPM is used by organizations to attest the health of a device. But if a system was compromised, it might report it was healthy, even when it wasn't.

This version had great success, it was introduced on most of the x86-based client PCs starting from 2005 and on most servers starting from 2008.

While the TPM 1.2 began to be widely used, the TCG started to work on the TPM version 2.0, represented in figure 2.2. This was necessary because of the publication, in 2005, of the first significant attack against the SHA-1 digest algorithm, which was heavily used in the TPM 1.2. TPM 2.0 implementation enables the same features of TPM 1.2, plus several more. One of the most significant improvements was the *Algorithm Agility*, which permits changing algorithms without revisiting the specification [5]. This feature was the first motivation for the design of TPM 2.0, but during the work, other features were added, such as support for the *Elliptic Curve Cryptography* algorithms, dedicated BIOS support, and others.

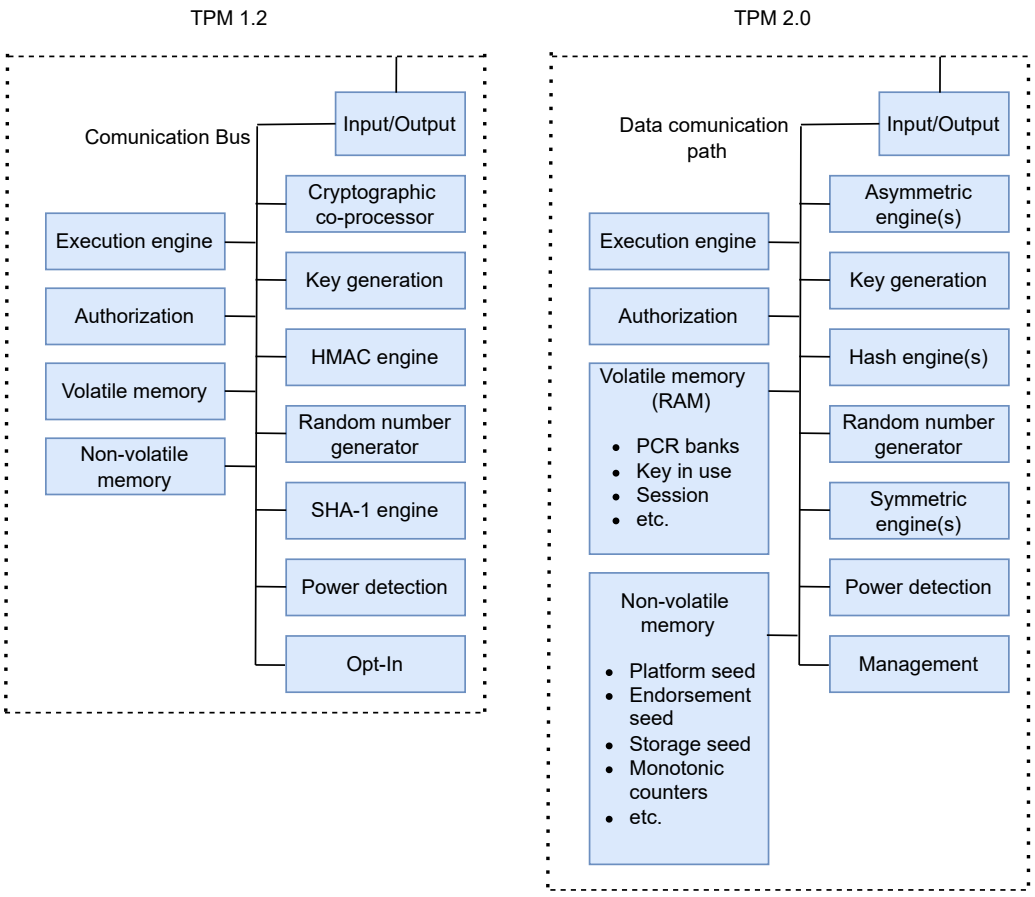

Figure 2.2. Comparison between TPM 1.2 and TPM 2.0 architecture (source: [5] [12])

2.2 Roots of Trust (RoTs)

TCG-defined methods rely on the Roots of Trust. This is a set of elements that has to be trusted because misbehavior cannot be detected. The aspect that a Root of Trust can't be checked if it is behaving properly or not, it is possible to know how it's implemented. For this reason, Certificates are used to prove that the root has been implemented in a trustworthy way. An example could be a certificate provided by an independent testing laboratory that may report the Evaluated Assurance Level (EAL) of a TPM, consequently providing confidence in the correct implementation of its RoTs.

In a Trusted Platform are required three Roots of Trust by the TCG:

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

2.2.1 Root of Trust for Storage (RTS)

The TPM memory is not accessible by any other component other than the TPM. Not all the information, contained in the TPM memory, is sensitive. Some non-sensitive information is not protected from disclosure, while other information is not accessible without the proper authentication (for example the private part of an asymmetric key).

2.2.2 Root of Trust for Measurement (RTM)

This component sends measurements to the RTS. Typically it is the CPU controlled by the Core Root of Trust for Measurement (CRTM), which is the first set of instructions executed when a new chain of trust is established. When the system is reset the CRTM is executed and sends values that indicate the identity of the RTS. It supports the integrity measurement of the Trusted Platform calculating digests on code and data and sending them to the RTS.

2.2.3 Root of Trust for Reporting (RTR)

This is the component that reports the content of the RTS. The report typically is a signed digest of the TPM content requested. The TPM doesn't report for every Shielded location, for example in the case of the private part of keys, the TPM will not report for that Shielded location. In general, values on which the RTR reports are platform state in PCRs, audit logs, and key properties (certify that an object with a specific name is loaded in the TPM, and guarantee that a public area, with a specific name, is associated with a corresponding sensitive area). The RTR is identified by asymmetric aliases (Endorsement keys) derived from a common seed (that should be unique for each TPM). The RTR needs to be bound with the platform because it's required the assurance that PCR values accurately represent the state of the platform. The proof of this binding can be a Platform Certificate.

2.3 Trusted Boot

A significant example of a technique to evaluate if a platform is in a trusted state is the *Trusted Boot*, in which each component executed during the boot process is measured and this value is stored in the TPM (RTS). This mechanism doesn't prohibit booting in an insecure state but permits registering the boot state and communicating it to an independent entity that will be able to verify if the system booted securely. This procedure is based on the *Transitive trust*, which exploits the trustiness of the RoTs and transfers it in a transitive way from an executable function to the next one that takes control over the machine.

The measure calculated on every boot component (code or data) is a digest performed with a cryptographic hash function and then stored in the TPM, more precisely in a PCR, which has the role of a shielded location of the RTS. The PCR's value can be modified with only two operations. the first one is the *Reset* operation, which is executed when the platform is powered on, and afterward only if the PCR has an attribute that allows the reset. The second one is the *Extend* operation which permits storing several hashes taking the old value contained in the PCR, concatenating it with the new measure and performing again the hash function on this string:

$$PCR_{new} = \mathbf{H}_{hashAlg}(PCR_{old} \mid\mid measure)$$

It is possible to store the whole sequence of boot in a single PCR, but for performance reasons typically are provided multiple PCRs for each module (the BIOS, the OS boot loader, ...).

In order to check the value contained in the PCR, if the boot sequence is known a priori, is sufficient to perform again the extension with the known values and confront the result with the value contained in the PCR. If the boot sequence is not known a priori it is possible to leverage the RTM that keeps a *Measurement Log* where is recorded each change of the system state (*Measurement Event*).

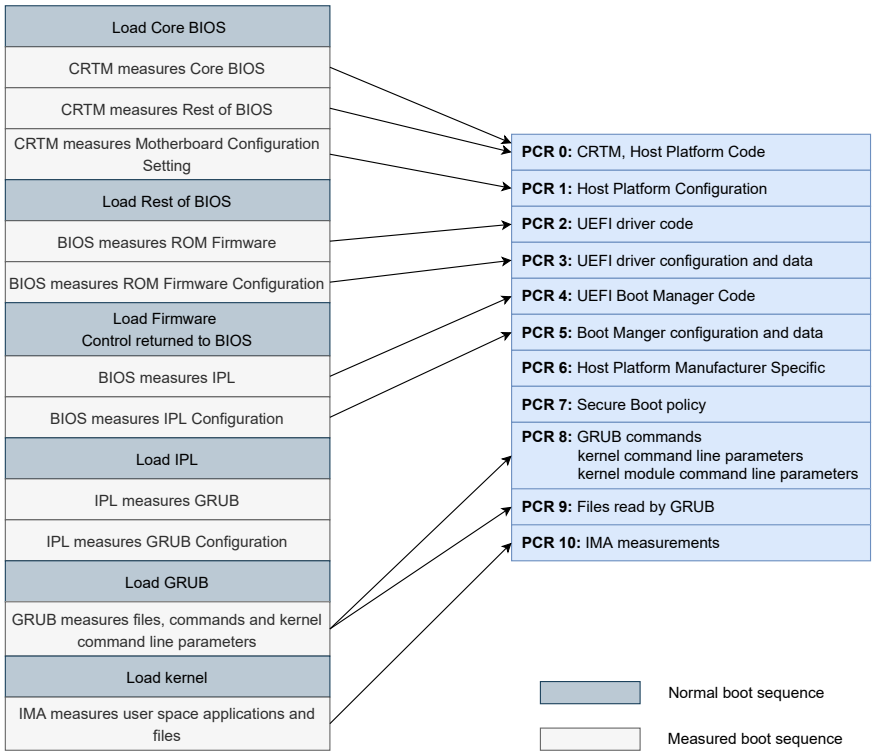

Figure 2.3. Trusted Boot (source: [10])

During the *Measured Boot*, described in figure 2.3, all software components invoked during the boot phase are measured and these values are stored in PCRs. Typically the first ten PCRs are reserved to record measures deriving from the boot process, the PCRs with an index higher than 9 are used to store values deriving from events after the boot phase. A possible description of the Measured Boot id that starts from the CRTM, a subset of the BIOS, that measures itself, the rest of the BIOS and extends the value in PCR 0. After that it measures the motherboard configuration settings and extends the value of PCR 1, then it passes the control to the rest of the BIOS, which measures the ROM Firmware Configuration and Data and extends the PCR 3, and then passes the control to the ROM Firmware. The control returns to the BIOS after the execution of the ROM Firmware which measures the Initial Program Loader (IPL), referred to as the primary boot loader, and extends the value of PCR 4, then it measures the IPL Configuration and Data and extends the value of PCR 5. After that the control passes to the IPL which load

the secondary boot loader (GRUB is the default one in x86 platforms) and passes the control to it, which extends the value of PCR 8 with all grub command executed, all parameters passed to the kernel and the modules of the kernel. It also extends the PCR 9 with any file it reads and then passes the control to the kernel.

Another kind of Measured Boot is the *Secure Boot*, which instead of just measuring boot components, checks them and if one of them has a measure different from the expected one, the boot is interrupted.

2.4 Remote Attestation Techniques

In several sectors, such as companies, industry, and critical infrastructure, Network Infrastructure (NI) is becoming more and more important. Those infrastructures could reach very large dimensions (figure 2.4), which make it almost impossible to attest the whole system, attesting one node at a time, and in case the integrity of a node would be compromised, could have a devasting effect on the whole infrastructure. *Remote Attestation* permits to attest integrity of a single node, which has to prove its trusted state to a remote verifier, but there are several proposals that try to aim for the attestation of more than one node. NIs are composed of many devices, of different kinds, like routers, switches, firewalls, and wireless access points, where each one of them has a specific software running on them. Moreover, in recent years there have been many research works regarding NIs to create higher-level abstractions [13], in this way it is possible to separate network functions from network devices using virtualization (*Network Function Virtualization*) and separate the network control from the actual network devices (*Software Defined Networking*). For all those reasons remote attestation become more and more important because even though devices are used in different ways, they still need to be protected from malicious users, so it is crucial to check their integrity.

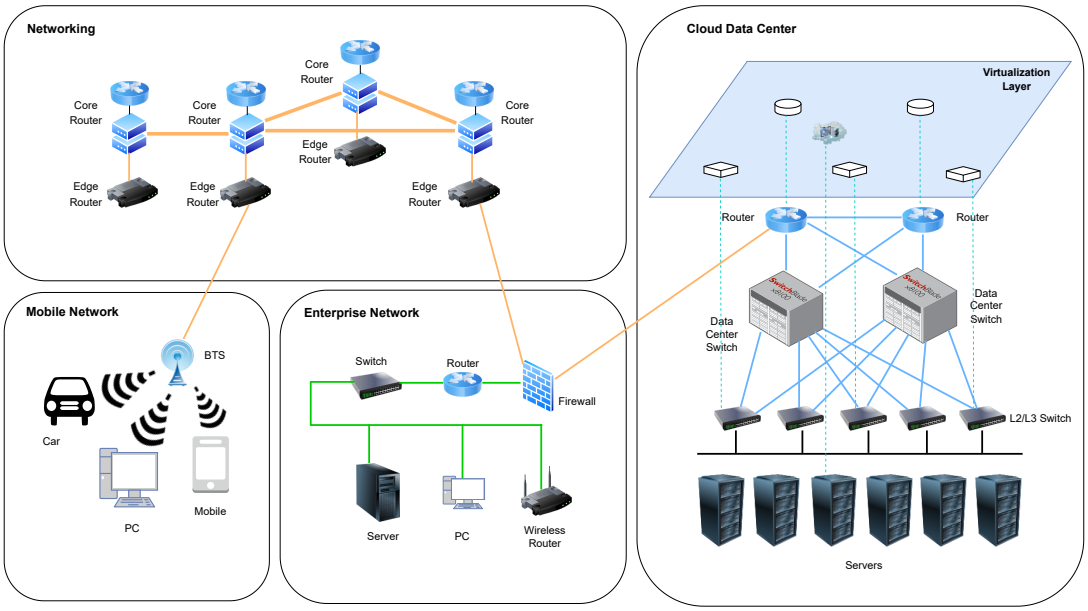

Figure 2.4. Network infrastructure deployed in several context (source: [13])

2.4.1 Remote Attestation

Remote attestation is a mechanism that enables a remote verifier to ascertain the integrity of a host that is in a known state. This mechanism follows a challenge-response protocol and the main entities in this protocol are the *Prover*(P) and the *Verifier*(V) [13]. The Prover has to send a

response that proves it was in an acceptable state at the time the attestation was requested by the Verifier.

The mechanism of attestation is typically based on a challenge-response protocol, where a trusted device (*Verifier*), verifies the integrity of an untrusted device (*Prover*). The process of attestation, as shown in figure 2.5, begins with the Verifier, which generally is assumed that knows in advance the correct state of the Prover, which challenges the Prover. At this point, the Prover has to compute the response, to demonstrate its trusted state. The untrusted device starts the attestation routine and computes the response based on the challenge received from the Verifier and its internal state. Once the Verifier receives this value, it can compare it with the known value, and if they match, the Verifier can assert that the Prover hasn't been compromised.

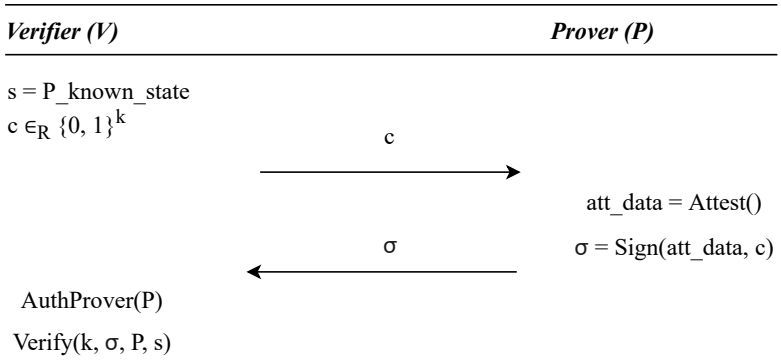

Figure 2.5. Attestation process. It is assumed the verifier and the prover have generated key pairs according to parameter k (source: [13])

Several attestation mechanisms have been proposed, but they are very different and use intrinsically different ways to achieve their goals [14]. For this reason, in general, all techniques are divided into three categories: *hardware-based*, *software-based*, and *hybrid* attestation.

2.4.2 Harware-based Attestation

Hardware-based techniques are based on a hardware device or component for realizing a remote attestation schema.

TPM

This device is one of the most significant examples of a tamper-resistant chip for remote attestation. As already reported before, the first version largely used was the TPM 1.2 and subsequently upgraded with the TPM 2.0. The TPM has some registers, where it stores hashes used for remote attestation, called Platform Configuration Registers (PCRs). The TPM 1.2 was based on the SHA-1 hash algorithm, which was substituted, in version 2.0, with the possibility to use more than one algorithm, making the design more agile and secure. Moreover, the TPM stores information (keys) regarding its identity, to be considered trusted. The first key enclosed in the TPM is the Endorsement Key (EK) and is generated during the manufacturing process. This key is used in almost all key-related operations in the TPM. Other keys that can be generated are the Attestation Identity Keys (AIK) used in digital signature operations and the storage keys used in encryption and decryption of data [13]. Several techniques have been proposed for supporting remote attestation with TPM 1.2, one of the first approaches used a Trusted Third Party (TTP) called Privacy-CA, which had problems with privacy [15]. It was used to perform the attestation of a single platform identity. To prove its identity, the platform uses the AIK certificate which is emitted by a trusted third-party Privacy CA. The problems related to privacy, derive from this procedure because in this way the Privacy CA will know the platform identity. Another problem is that this Privacy CA it's a single point of failure, which can be a bottleneck in case the verifier queries it at each attestation. Another schema proposed was the Direc Anonymus

Attestation (DAA), which can be seen as a group signature without the feature that a signature can be opened, i.e., the anonymity is not revocable. In addition, DAA allows for pseudonyms, i.e., for each signature a user can decide whether or not the signature should be linkable to another signature [16].

Direct Anonymus Attestation

Another schema proposed was the Direc Anonymus Attestation (DAA), which can be seen as a group signature without the feature that a signature can be opened, i.e., the anonymity is not revocable. In addition, DAA allows for pseudonyms, i.e., for each signature a user can decide whether or not the signature should be linkable to another signature [16]. This schema includes three entities:

- *Platform*: It is a system composed of a host computer and a TPM, and both components participate to create a signature that can prove to the verifier that the platform itself was indeed issued a credential by the issuer. In this way, the verifier is not able to know the platform identity
- *Issuer*: Its main role is issuing credentials to platforms and checking a revocation list to know if a platform is compromised and if so, revoke it.
- *Verifier*: It has the purpose to verify it the platform is compromised or not.

This schema uses zero-knowledge proof to attest that the remote attestation originates from an authentic TPM without disclosing to the verifier the exact identity of the TPM. The issuer in DAA is provided with an anonymous credential instead of an identity for the TPM. Then the TPM proves its identity to the verifier using zero-knowledge proofs and its pseudonym [13].

Binary Remote Attestation

In this case, PCRs represent the state of the platform and the TPM has to sign values contained in PCRs. After this process, the signature is sent to the verifier together with the measurement log, which is the list of all files and binaries, with the relative hash, in execution on the platform. Once the verifier receives all this data, it can verify the current state of the platform. Examples of realization of this technique are *IBM Integrity Measurement Architecture* (IMA) [17] and the *Trusted Linux Client* (TLC), which permits to protect desktop and mobile Linux clients from online and off-line integrity attacks [18]. Some disadvantages of this schema are that the configuration of the software and the hardware are disclosed to the verifier and a minimal change in software or their configuration could create problems for the integrity evaluation.

Property-based Attestation (PBA)

This technique permits the attestation of the platform configuration without disclosing it The check is performed on the *properties* which are the mapping of the platform configuration.

Physical Unclonable Functions (PUFs)

A *Physical Unclonable Function* (PUF) is a noisy function that is embedded int a physical object [19]. These hardware-based security primitives can be used for the construction of a hardware component that can protect the attestation process.

2.4.3 Software-based Attestation

Hardware-based attestation is a very good solution for remote attestation, but it is not always suitable, because it needs hardware and software resources that may not always be available, for example in embedded devices. For this reason, some software-only remote attestation approaches have been proposed, to remedy the hardware overhead.

An example of a software-based primitive is *Pioneer* [20] which is one of the first approaches to remote attestation without being based on CPU-architecture extension or any secure co-processor. This solution is based on a challenge-response protocol (figure 2.6) between two entities: the first one is an external trusted entity called *dispatcher*, and the second one is an untrusted entity called *untrusted platform*. The key concept of this technique is that the dispatcher uses Pioneer to dynamically build a trusted computing base (*dynamic root of trust*) on the untrusted platform, and once established is guaranteed that all the code contained is unmodified. The component which permits the instantiation of this dynamic root of trust is the *verification function*.

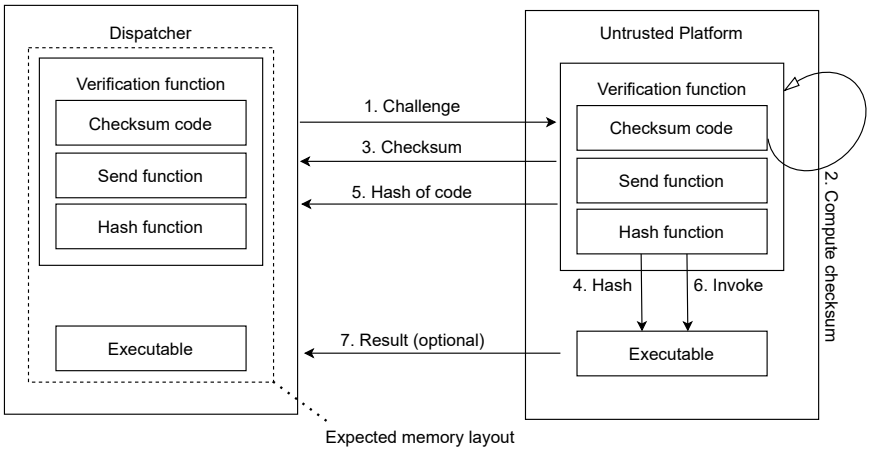

Figure 2.6. Schematic representation of how Pioneer works (source: [20])

This component is the main one of Pioneer because is the one designated to perform an integrity measurement on the executable, setting up an environment that ensures untampered execution, and it has three parts:

- *Checksum code*: This part is used to create an untampered execution environment for the secure execution of the hash function, the send function, and the executable, and it has to compute the checksum on the whole verification function, which is like a fingerprint. In this way, a correct checksum will prove to the dispatcher that the verification function hasn't been modified. Some malicious users could try to manipulate the checksum forcing it to be the correct one, even if the verification function has been modified. In this case, it can be possible to detect this behavior because the verification function is made in such a way that if someone were to try to manipulate the checksum this computation would take much longer than the normal computation, and therefore you might notice the manipulation.

- *Hash function*: To calculate the integrity measurement of the executable it is used the SHA-1 hash algorithm. Even if the collision resistance of this algorithm has been compromised, it is used because of the second-preimage property, for which SHA-1 is still considered secure. For this reason, the has is not calculated only on over the executable, because in this case, an attacker could be able to find another executable that would produce the same hash. The measurement of the executable is performed as a function of a nonce sent by the dispatcher, and after that, the hash function invokes the executable.

- *Send function*: This function sends the checksum and the integrity measurement to the dispatcher.

The main idea behind Pioneer is to create a special checksum function with run-time side-effects for attestation, in this way any malicious manipulation of this function can be detected through additional timing overhead incurred from the absence of those side-effects [21].

2.4.4 Hybrid Attestation

The main shortcoming of the software-based approach is that it makes strong assumptions about adversarial capabilities, which may not hold in practical networked settings [21]. A hybrid approach consists of a software-hardware design. One example is SMART [22]: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. This approach is based on a *minimal* hardware modification of actual embedded Micro Controller Units (MCUs), which represents the first minimal hardware solution for establishing a dynamic root of trust in embedded devices. This primitive has two characters, a *Prover* (P), which will be the entity to attest, and a *Verifier* (V), which will be the entity that will have the duty to verify the integrity of the prover. As already reported many times, the basic schema consists of the prover which attests a specific region of code and sends a proof of execution to the verifier. A characteristic of SMART is that it guarantees the execution of a piece of code even if the prover is totally compromised.

Like many remote attestation technologies, SMART is based on a challenge-response protocol, started by the verifier (V), that exploits some specific hardware resources of the prover (P), and the basic functioning is shown in figure 2.7. The protocol starts with V that computes a *nonce*, and sends it to P, with other values: a and b which are the boudries of the region of code to attest; x which is the address of the code to execute after measurement process only if x_{flag} is setted. The value C is computed with a ROM-resident code on P, which calculates a checksum on the nonce received and the region between a and b, and, if x_{flag} is set, pass the control to x, and terminating the execution of this piece of code P send C to V. The checksum is computed using an HMAC algorithm and a key K, which is stored in a secure portion of the P's MCU, and the access to K is restricted so that only the code used to compute the checksum can use it. V to verify the value received from P will compute a new value with the same parameters and K.

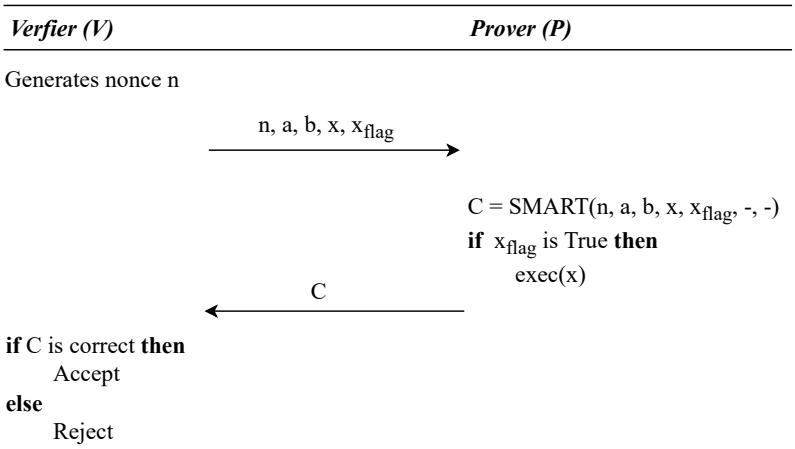

Figure 2.7. SMART overview (source: [22])

2.5 Trusted Execution Environment (TEE) overview

The Trusted Execution Environment (TEE) technology has been developed for being able to execute sensitive code, and read data, in a protected way. Several CPU vendors have developed their versions (e.g. Intel SGX, ARM TrustZone, and AMD SEV). The secure execution environment is typically called *enclave*. Unfortunately, each vendor TEE enables only a small subset of the possible design space across threat models, hardware requirements, resource management, porting effort, and feature compatibility. Recently it has been proposed some open-source frameworks

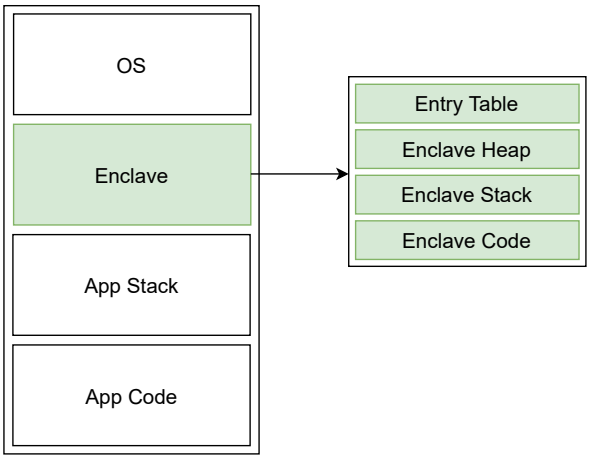

Figure 2.8. Enclave within application's virtual address space (source: SGX: the good, the bad and the downright ugly)

like *Keystone* [23] that work without hardware modification but use abstractions provided by the hardware like memory isolation and programmable layers under untrusted components (e.g. OS).

2.5.1 Intel SGX

Intel Software Guard Extension (SGX) is an extension of the assembly language of the Intel architecture regarding instructions and memory access, which allows an application to instantiate a protected container called *enclave*.

An *enclave* is a separated and encrypted memory zone for code and data, figure 2.8, in application's address space. This region provides confidentiality and integrity, and only the code resident in the enclave can access it, all other requests are prevented, even from privileged software like operating systems or BIOS [24]. The enclave is stored in a subset of DRAM, called *Processor Reserved Memory* (PRM), which is a continuous memory region with a size that is a power of two and the start address must be aligned to the same power of two.

The enclave's content and the associated data structures are stored in the *Enclave Page Cache* (EPC), which is a subset of the PRM [25]. An enclave is created and initialized by untrusted software, and the hardware assures that the enclave can be modified only before the initialization. During the initialization phase, the enclave is measured, and the value measured will be used for local and remote attestation.

Intel has modified the hardware memory controller: since the only way to read/write on RAM is to go through the controller, then this modified controller can block access from the outside. Encrypting everything is not a solution because if the actual enclave's code must be executed, and everything is encrypted in the RAM, what happens is that you need to decrypt the code, which is done by the OS, before starting the scheduling. By doing this, the code is stored decrypted in the RAM and there is no assurance. So, this is why it's needed to have the enclave's code and data unencrypted in RAM but in a way where only the enclave's code can access the enclave's data in RAM. And this is why the memory controller actually denies everyone to access this memory. The structure of the memory is represented in figure 2.9. Inside the PRM there is the EPC. Inside the EPC there will be the OS paging. This is compatible with the existing paging scheme since the idea is to have retro compatibility. There will be pages of 4 KB that will contain the enclave's code and data. Then, in another part of the RAM, there will be the *Enclave Page Cache Map* (EPCM), which is an array with one entry per EPC page, so computing the address of a page's EPCM entry only requires a bitwise shift operation and an addition [25]. Each entry will contain three sub-entries: the first one is a *VALID* bit, which permits understanding if the page is valid or not, where valid means if the page contains the enclave's data and code of an active enclave. The second one is the *Page Type* (TP) used to distinguish if the page is assigned to a normal

software enclave or an enclave used by the SGX implementation. The last one is *ENCLAVESECS* which is a structure that identifies the enclave that owns the page, so it is a reference to another structure that identifies an enclave.

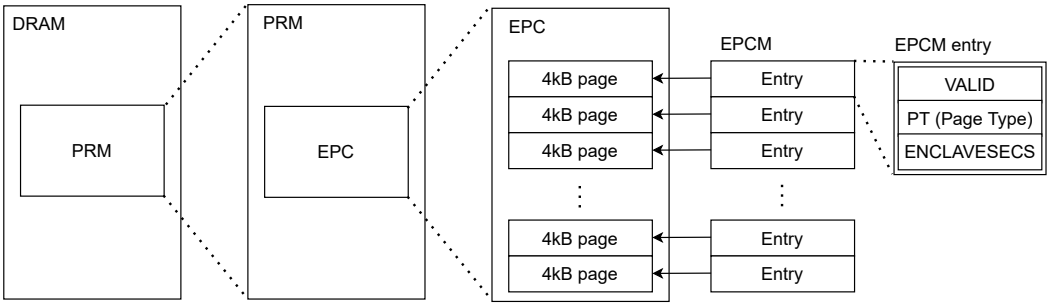

Figure 2.9. PRM structure (source: [25])

SGX permits both the local and the remote attestation. The base process for the remote attestation is the *Enclave Measurement*, which identifies the software running in the enclave. It is computed with 256-*bit SHA-2 secure hash* function, and this hash is stored in the enclave's *SECS* (*MRENCLAVE* field). This value is computed by the OS during the enclave initialization in the following way:

1. *ECREATE*: It executes the SHA-2 initialization algorithm, so goes into the SECS, in MRENCLAVE, and inserts the results of the initialization algorithm;
2. *EADD*: It hashes the enclave virtual address + a structure that contains the security attributes relative to the page (*SECINFO*), in fact, this is what goes inside the EPCM (metadata of the page in the enclave page cache map);
3. *EEXTEND*: It hashes 256 bytes of the enclave data (must be called multiple times);
4. *EINIT*: It is the SHA-2 finalization algorithm, that computes the valid final hash, which will be inserted in MRENCLAVE (valid enclave measurement).

The **software attestation phase** begins with an *attestation challenge* made by the user, then the *EREPORT* instruction is called, and it takes the Measurement of the enclave and the signer to create a report. The report is signed with a unique key (inside the CPU). This report, before being sent to the user, goes to the *SGX Quoting Enclave* which checks the signature and reauthorizes the report. Then the *attestation signature* is sent to the user.

To make everything work, there is a shared secret between the CPU e-fuses (one-time programmable part of the CPU that is tamper-resistant) and the Intel Provisioning Service (which checks the signature). The provisioning Enclave, which runs inside the untrusted computer, obtains an attestation private key from the Intel Provisioning Service (IPS). This private key is stored encrypted in DRAM. Then, the remote party challenges the application enclave, the application enclave calls EREPORT instruction, the quoting Enclave verifies the report locally, and then, if the verification is valid, it sends the report to the remote party signed with the attestation private key. In this way, the remote party knows that the remote computer is running a correct SGX implementation because the remote party asks the IPS for the attestation public key and verifies the report's signature.

Chapter 3

Remote attestation standards and use cases

3.1 Device Identifier Composition Engine (DICE)

The specification *Device Identifier Composition Engine* (DICE) [26] provides new basis for remote attestation specifically suitable for IoT devices. This was necessary because it could be needed remote attestation of devices' identity and firmware integrity of IoT devices that cannot have a Trusted Platform Module attached cause the limited resources. DICE-based approaches permit the establishment of an IoT secure environment without modifying the hardware of already deployed devices.

In industrial and academic contexts several techniques for remote attestation of devices with limited resources have been developed. For example, an industrial solution is TrustZone by ARM which establishes a Trusted Execution Environment (TEE) that permits the integrity firmware attestation. Academic solutions are SMART [22] or Sancus [27] that expand existing MCUs' architecture to establish secure attestation schemes. Unfortunately, none of the mentioned solutions is usable for devices based on older MCUs.

The DICE solution was created by the Trusted Computing Group (TCG) in order to support the presence or the absence of a TPM. This technique permits to derive the cryptographic identity of a device from its firmware and a Unique Device Secret (UDS), in this way is possible to derive keys for attestation and other purposes. DICE's purpose is to anchor a Root of Trust in devices with minimal hardware overhead. For this reason, the necessary resources are kept as minimal as possible [28]:

- Read-only boot code is possible, preferably on time programmable;
- A UDS of at least 128 bits can be stored on the device. A one-time programmable UDS is preferred;
- Only the boot code may access the UDS. A lockout mechanism prevents the firmware from accessing it.

It is implemented as the first code executed by the device after it's powered on, deriving the cryptographic identity, called Compound Device Identifier (CDI), from the UDS and the measure of the first mutable code, calculated with a one-way function. The CDI, once calculated, has to be delivered to the first mutable code. In this way, it will be possible to derive keys from the CDI, and those keys will be bound to the UDS and the firmware, so a change in one of them would be detected because it would result in a different CDI and so different keys. The basic architecture is two-layered, as shown in figure 3.1, which is based on the first layer, DICE, and the second layer, mutable Firmware. One large disadvantage of this architecture is that if there is a firmware update, the device's identity derived from the CDI will change.

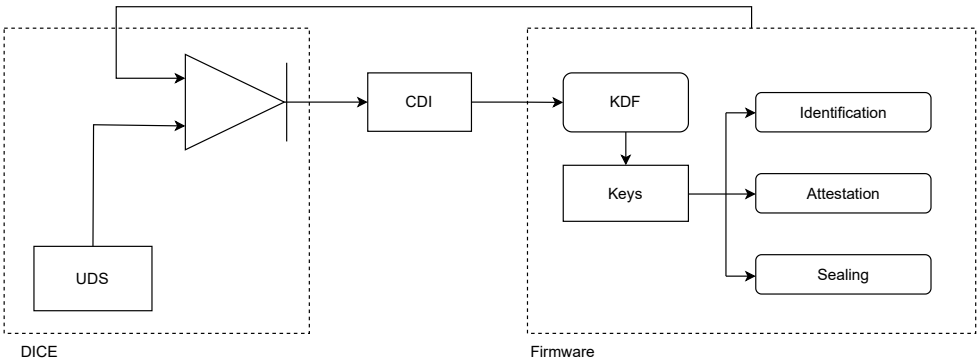

Figure 3.1. DICE two-layered architecture (source: [28])

A solution to this problem is the RIoT [29] architecture, where firmware consists in n layers and each layer i is provided with its key $K_{L,i}$ which derives and it is correct only is the key $K_{L,i-1}$ is correct. In this case, it could be possible to divide the firmware into multiple layers, so the first mutable code derives the identification key (K_{ID}) from the CDI. After that, it can provide the second mutable code with a key $K_{L,2}$ derived from the second mutable code and the CDI, and then hand both (K_{ID}) and $K_{L,2}$ over to the second mutable code. For this solution to work, it's needed that the first mutable code is designed as bug-free as possible in order to remain on the device for its whole lifetime. In this context, the UDS is the same of $K_{L,0}$ and the CDI is the same of $K_{L,1}$ An example of how DICE and RIoT can be integrated into a system is shown in figure 3.2.

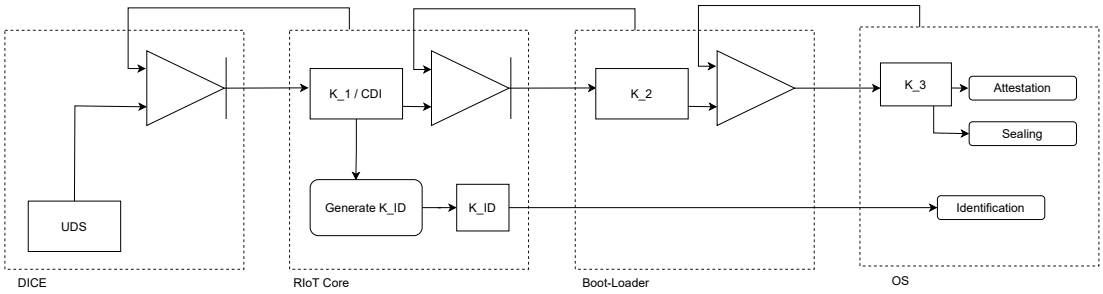

Figure 3.2. DICE in a RIoT architecture with three layers (source: [28])

For remote attestation in [29] are reported two schemes:

1. *Asymmetric Cryptography*: This solution is based on the derivation of an asymmetric key pair (K_{ID}) from the CDI. After that, it is randomly generated another key pair for the second mutable code ($K_{L,2}$), and a measurement of the second mutable code is concatenated with its public key and everything is signed with $K_{ID,private}$. Once completed this operation, the signature, $K_{ID,public}$ and the key pair $K_{L,2}$ are passed to the second mutable code, in this way the device can check the integrity of the second mutable code and identify itself, using the signature, $K_{L,2,public}$, and $K_{ID,public}$.

2. *Symmetric Cryptography*: This scheme is based on a shared secret (S_{att}), between the attestation server and the device, which is stored on the device encrypted with $K_{ID,public}$. In this way is possible to use a proof-of-possession attestation protocol, being able to use symmetric cryptography.

The first scheme, using asymmetric cryptography, has many problems from the performance point of view, especially on very small IoT devices. The second one is more lightweight than

the first one because of the symmetric cryptography, but it still needs asymmetric cryptography for the encryption of the shared secret. A possible solution to avoid this issue is using MACs to protect the authenticity and integrity of data. As already mentioned before, the CDI is computed with the UDS, and a measurement of the first mutable code, and it is derived correctly only if these data are correct. For this reason, the CDI could be used as a way for checking the device's identity and firmware integrity. For this purpose, using a one-way key derivation function, are calculated two symmetric keys: K_{ATT} for attestation, and K_{ID} for device identification. This mechanism works because K_{ATT} is derived correctly only if CDI was derived correctly, which proves the correctness of UDS, so it is possible to prove the correctness of CDI and the integrity of the first mutable code using it in a proof-of-possession protocol for attestation to the server. For this solution, it is assumed that the device has a structure with DICE and mutable firmware, but it can be easily implemented on a multi-layer-firmware. An example of how this scheme could work is shown in figure 3.3, where the server sends a *nonce* (N) to the device, which computes a MAC (R) on N and K_{ATT}. This operation is made also by the server that computes R' using its copy of K_{ATT}. Then the device sends R to the server which will confront it with R'.

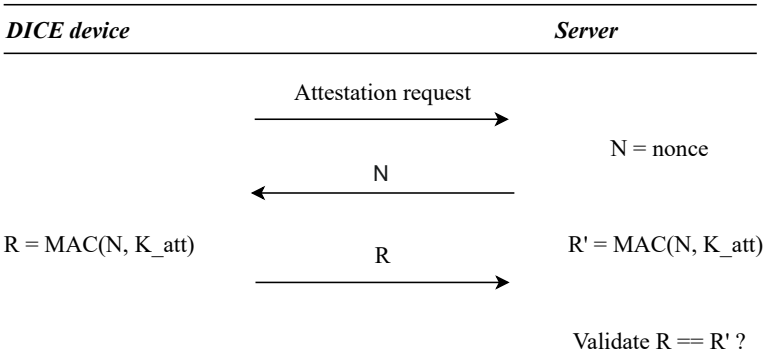

Figure 3.3. MAC-based attestation protocol (source: [28])

3.2 Remote Attestation Procedures (RATS)

Remote **AT**testation Procedure**S** (RATS) architecture permits in an easier way the attestation of devices' characteristics generally based on the trustworthiness of another specific device. Generally, attestation is based on messages that communicate trustworthiness properties associated with a specific Attested Environment (*Evidence*). This Evidence is compared with the reference's values, which typically consist of firmware or software digests and some information that explains why the attesting module is a trusted source of Evidence. After this operation will be produced Attestation Results that help Relying Parties to determine levels of trust [30].

In RATS architecture are introduced several roles and relationships between them. These roles, shown in figure 3.4, are assigned to *entites* which typically are system components, such as devices [31]:

- *Attester*. The purpose of this role is to create Evidence that will be transferred to the Verifier in order to be checked.
- *Verifier*. The entity to which this role has been assigned, uses the Evidence, any Reference Values from Reference Value Providers, and any Endorsements from Endorsers, applying an Appraisal Policy for Evidence in order to be able to verify the trustworthiness of the Attester. After that, the Verifier generates the Attestation Results.
- *Verifier Owner*. It is typically assigned to an administrative entity, which has the authorization to configure Appraisal Policy for Evidence in a Verifier.

- *Appraisal Policy for Evidence.* There are several methods in which the Verifier can have this information. The first one is that this data is provided by the Verifier Owner, or can be configured on the Verifier by the Verifier Owner. Another possibility is that the Appraisal Policy for Evidence is programmed into the Verifier, but it can be obtained with some other mechanism.
- *Relying Party.* This role is assigned to the entity which performs the appraisal of Attestation Results. This operation consists in using the Attestation Result to make some decisions, related to the specific application.
- *Relying Party Owner.* This role has the same characteristics as the Verifier Owner, but in this case for the Appraisal Policy for Attestation Results, having the authorization to configure it in a Relying Party.
- *Appraisal Policy for Attestation Results.* It can be obtained in a similar mechanism to Appraisal Policy for Evidence. In this case, it can be programmed on the Relying Party, it can be obtained from the Relying Party Owner, it can be configured on the Relying Party by the Relying Party Owner or in some other way.
- *Endorser.* The entity which has this role can help Verifiers to check the authenticity of Evidence with its Endorsements. In this way, Verifiers can also deduce further capabilities of the Attester.
- *Reference Value Provider.* This role is typically assigned to a manufacturer entity and its Reference Values help Verifiers to check Evidence in order to decide if acceptable known Claims have been recorded by the Attester.

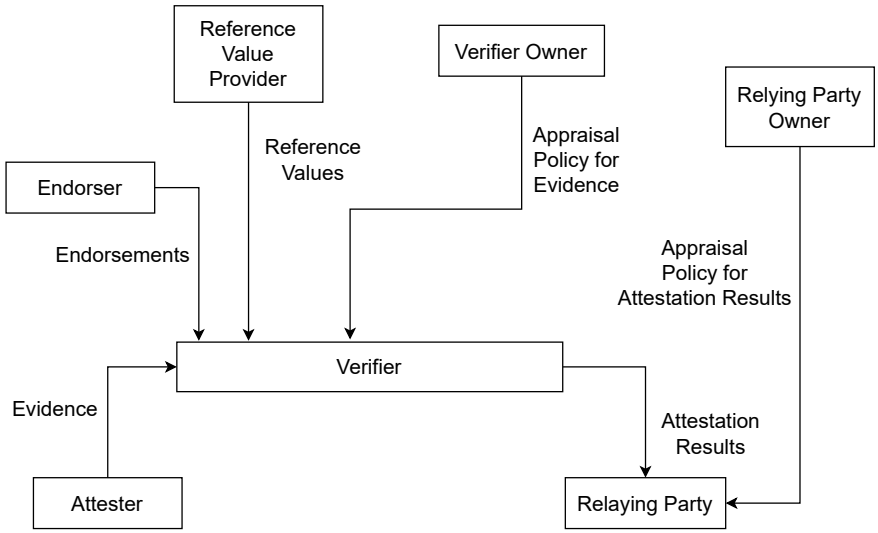

Figure 3.4. RATS's roles overview (source: [31])

In figure 3.4 are reported also Conceptual Messages exchanged between entities:

- *Evidence.* It is a set of Claims about a specific environment that with them, reveals its operational status. This information is used by a Verifier to determine its relevance, compliance, and timeliness. The Evidence has to be associated in a secure way with the Attester, in order that the Verifier is able to understand if Claims originated from a different environment. In addition, Claims have to be collected in some way that the Attester cannot lie about its trustworthiness.
- *Endorsements.* This object is a declaration that some entity guarantees the integrity of several functionalities of the device. An example could be the signing capability in hardware. In this case, the endorsement can be the manufacturer certificate that signs a public key

where the corresponding private key is accessible only inside the device's hardware. Using Endorsement together with Evidence permits to run an appraisal procedure, based on appraisal policies. These policies can be specified for the device or specific to the manufacturer which provides the Endorsement.

- *Reference Values.* These are values used by a Verifier to compare to Evidence provided by an Attester. They are provided by a Reference Value Provider and if they match the Evidence that means acceptable Claims, also based on appraisal Policy.
- *Attestation Results.* They are the information used by the Relying Party to decide if a particular Attester can be considered trusted or not. The structure of Attestation Results can contain a single boolean that permits to evaluate the trustworthiness of the Attester or can be more complex, containing a larger set of Claims. An Attester is considered not compliant by default by the Relying Party; it can be considered so by the Verifier, after some analysis that contemplates Appraisal Policy for Attestation Results and the information provided by Attestation Result.
- *Appraisal Policies.* These policies specified some constraints that will be used from the Verifier when apprising Evidence, and the Relying Party when apprising Attestation Results.

3.2.1 Topological Pattern

In [31] are reported two reference models that reifne the data-flow shown in figure 3.4:

Passport Model

This model is conceptually very similar to how passports are issued to citizens. It is possible to make an analogy by thinking of the citizen as the Attester, the Passport issuing agency as the Verifier, the passport application as the Evidence, the passport as the Attestation Result, and the immigration desk as the Relying Party. The logical flow of this model (figure 3.5) is that the Attester produces the Evidence and sends it to the Verifier. At this point, the Verifier compares the Evidence against its appraisal policy and produces the Attestation Result which will be sent back to the Attester. The Attester stores this information and then it can send to a Relying Party which will compare it with its appraisal policy, in order to be able to make a decision.

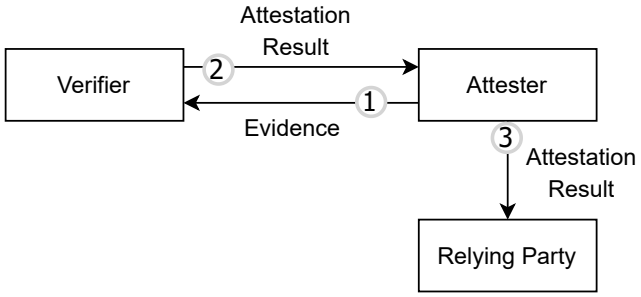

Figure 3.5. Passport Model (source: [31])

Background-Check Model

This model is a bit different from the previous one, in fact, in this case, an Attester produces the Evidence and then sends it to a Relying Party. The Relying Party treats the Evidence received as a blob, and it forwards to a Verifier. The Verifier, like in the previous case, confronts the Evidence with its appraisal policy and produces the Attestation Result. Then the Verifier sends the Attestation Result to the Relying Party which confronts it with its own appraisal policy.

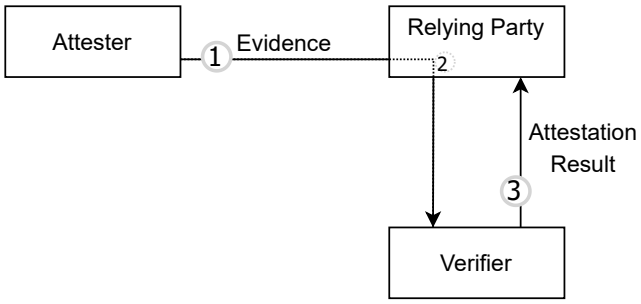

Figure 3.6. Background-check Model (source: [31])

3.3 Trusted Execution Environment Provisioning (TEEP)

Some critical applications need some protection during their execution in order to not compromise them or to not disclose data that they are working on. In many contexts, an application is executed in a system that can run other software in parallel, so if an application needs to be protected, should be isolated in some way from the others. As explained in the previous chapter, the Trusted Execution Environment (TEE) concept can be a solution to protect critical code or data, because it permits the execution of an application in a protected environment that prevents code to be tampered with or data to be read outside that environment. Because the fact there are several TEEs, because different vendors have proposed their own solutions, it's needed some protocol that helps developers and service providers to manage *Trusted Applications* (TAs) running in different TEEs. This protocol would permit several possibilities [32]:

- The first possibility is that a Service Provider that intends to provide a service through a TA, needs to check the device on which will run the TA, so, for example, it needs to verify the root of trust of the device or the type of TEE offered by the device;
- On the other hand a TEE has to verify if a Service Provider that wants to execute TAs in the device is authorized to manage TAs in the TEE;
- All the parties that take part in the protocol need to be able to verify that a TEE can provide the security required by a TA;
- A Service Provider must have the capability to understand if a TA is installed on a device and if not it can install on its TEE. In addition, it has to be able to get the version of the TA too, and update it if it is not the latest version available;
- A Service Provider must be able to remove a TA in a TEE depending on the fact that this TA is still offered or not, or maybe has been revoked for some reason;
- A Service Provider must have the ability to define relationships between TAs under its control in order to decide in which way they have to communicate and share data or keys.

This approach can be used in several scenarios, for example in *electronic payments*, where its needed high security and trust in the hosting device (e.g. mobile device). In this case, could be needed some biometric information that can be stored in a TEE. Another possible use case is *confidential cloud computing*. In this case, a tenant could have sensitive data, such as credit card numbers to manage, and can store them in a TEE. In this way the tenant will be the only one that can access that data, so neither the cloud hosting provider will be able to access this data.

3.3.1 Architecture

In figure 3.7 is represented the set of components typically present in a device with TEE [33]. Starting from the outside components, *Trusted Component Signer* and *Device Administrators*

mage TAs running on devices using services made available by a *Trusted Application Manager* (TAM) permitting a remote administration. A TAM is a component responsible for the management of Trusted Components, which are code and data in a Tee, under the instructions of the Trusted Component Signer and Device Administrator. The management that is done by the TAM occurs because of the interactions that have with devices' *TEEP Broker* which relays communication and messages between a TAM and a *TEEP Agent* that are executed in the TEE. A TAM can be publically available, in order to be reached by Trusted Component Signers and Device Administrators, or can be private in order to limit entities that can access it. The TEEP Broker, as already reported manage communication between a TAM and a TEEP Agent. It runs in a Rich Execution Environment (REE), so outside the TEE. The *TEEP Agent* runs inside the TEE and receives requests of a TAM through the TEEP Broker, and it can parse or forward these requests, depending on the TEE provider's implementation. In any case, it will send a response to the TAM, again passing through the TEEP Broker.

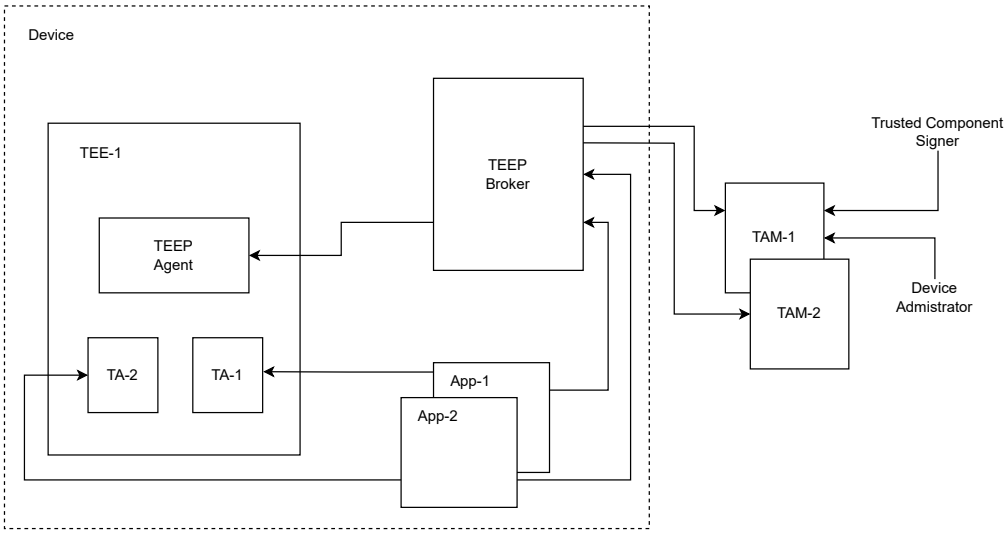

Figure 3.7. TEEP Architecture (source: [33])

In some cases is possible that a device implements more than one TEE. In these cases, it could be possible having one TEEP Broker that manages all TEEs, but for some TEE implementations this solution doesn't work (e.g. Intel SGX), so it would need a TEEP Broker for each TEE.

3.4 ETSI GR NFV-SEC 018

3.4.1 Network Functions Virtualization

In traditional networks, in order to implement all the Network Functions (NFs), such as Firewall, IDS, and Network Monitor, it is needed an important number of dedicated hardware. To offer a more flexible service an Internet Service Provider (ISP) offers the possibility to install these special-purpose apparatuses which the network administrator has the duty to manually configure. Unfortunately, this approach causes an important waste of resources and time, because of the continuously growing demand of customers.

A solution proposed to manage all these problems is the *Network Functions Virtualization* (NFV) paradigm. This innovative network paradigm permits to substitute the traditional network components that implement NFs, with a virtualized infrastructure that permits to deploy NFs as *Virtualized Network Functions* (VNFs). They are a software implementation of physical components that run inside containers or virtual machines. This approach has several advantages: it consents to separate NFs from hardware dependencies decreasing the deploying, configuration, and managing time. In fact, now VNFs run on servers, and the virtualization enables autoscaling

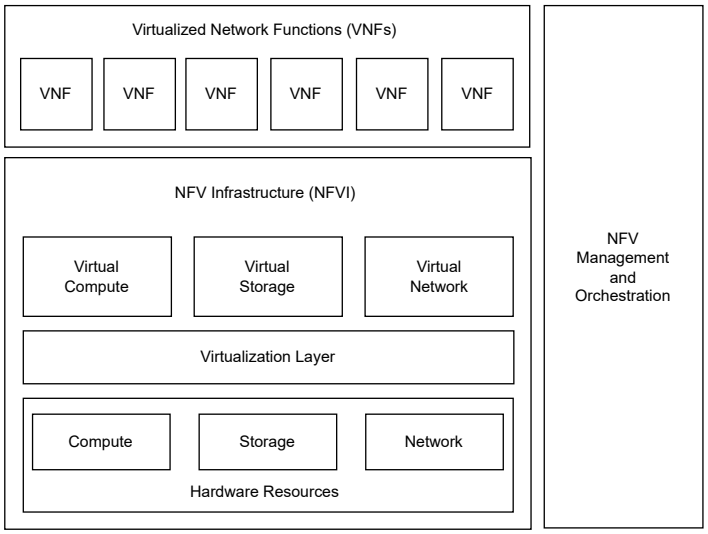

Figure 3.8. High-level NFV framework (source: [34])

of components which could be *Vertical Scaling* (add more resources to the single NF) or *Horizontal Scaling* (increase the number of VMs running that implement the same NF and add a load balancer in order to split the incoming traffic).

In order to allow the integration and the possibility for several stakeholders to deploy NFV in their architecture, the "European Telecommunications Standards Institute" (ETSI) established an "Industry Specification Group" (ISG) which works on the standardization of the NFV technology [34]. As shown in figure 3.8, an NFV framework is based on three main components:

- *Virtualized Network Function*, it is the network implementation of a network function, that can run in an NFVI;
- *NFVI Infrastructure* (NFVI), it is the set of infrastructure physical components, virtualized components, and technologies that allow Virtualization;
- *NFV Management and Orchestration* (MANO), it is the block that manages the lifecycle and orchestration of physical and software resources that enable the virtualization in the infrastructure, and it manages the lifecycle of VNFs.

The NFV paradigm, as already mentioned, introduces several advantages, but it introduced also some issues from the security point of view. For this reason, ETSI created the NFV-SEC Working Group (WG) which defines the set of possible threats of NFV, which include generic threats of virtualization, like memory leakage or interrupt isolation, and also threats of physical NF, like flooding attacks or routing-related security issues. Because all these problems become more and more important the monitoring activity in an NFV scenario because all VNFs are software implementations so they are vulnerable to manipulation from malicious actors.

To address all of these problems Remote Attestation can be used, to verify the trustworthiness of NFVs and resources that enable virtualization. ETSI proposed a document [6] (ETSI GR NFV-SEC 018) that identifies Remote Attestation architecture usable for NFV systems. The attestation scope, represented in figure 3.9, includes Hypervisor, which implements virtualization techniques, VMs, that run on top of the Hypervisor, and application processes running inside VMs. This permits to achieve the assurance that the service exposed is trusted.

3.4.2 NFV Remote Attestation Architecture

In a context of non-virtualization, remote attestation typically consists of a process that involves a *System under Evaluation* (SuE), which is the attester, and the *Remote Attestation Server* (RAS), which is the verifier. This process is characterized by three phases [6]:

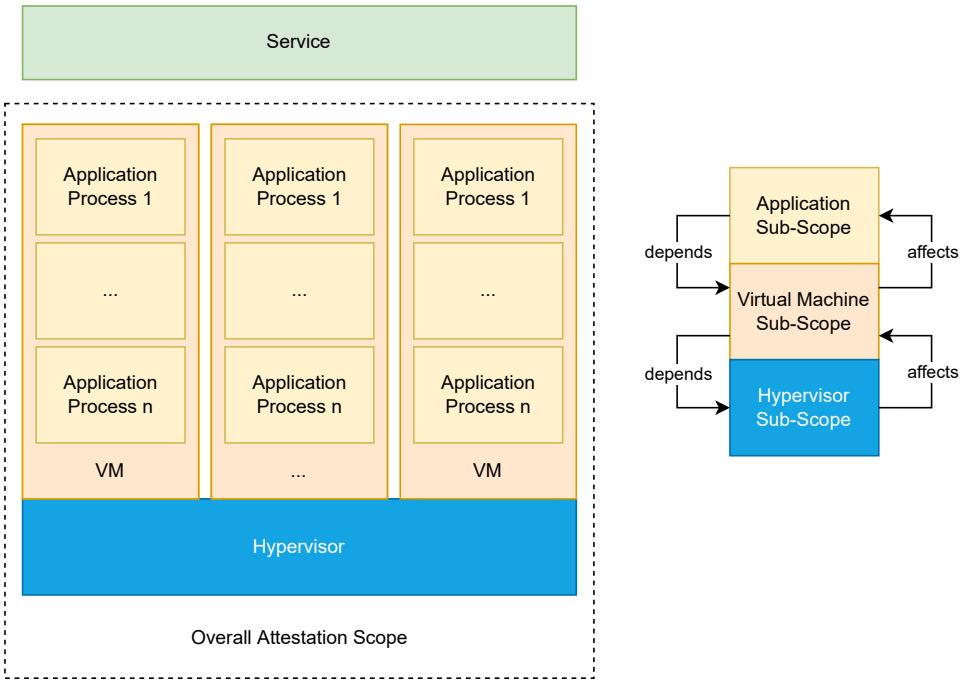

Figure 3.9. NFV attestation scope (source: [6])

1. ***Measurement***, that is the phase where the SuE accumulates all the needed information to check if is in a trustworthy state or not. To do so it is executed a measurement function that collects all the required data, maintains it in the Measurement List, and securely stores it.

2. ***Reporting***, is the phase executed once all evidence of the SuE is stored. A reporting function is executed which takes the integrity information with the Measurement List and creates a Report to send to the RAS assuring some security properties, such as integrity, authenticity, confidentiality, and replay protection.

3. ***Verification***, is the last phase executed once the RAS receives the report from the SuE. At this point, the RAS runs a verification function that confronts values received in the report with reference values already present in the RAS. After this computation, the verifier will produce a Result that contains the details about the trustworthiness of the SuE.

In the NFV context, for evaluating the trustworthiness of a network service, it's needed to evaluate all components that are "under" the network service, so every element on which the service depends. To achieve so, the remote attestation of an NFV has to start from the hypervisor-platform, in order that if it is trusted, the physical hardware, the hypervisor's software, and eventually the virtualized hardware can be considered trusted. Once NFVI layers have been attested, the next step is checking VNF layers that include all VMs' software stacks and all packages that implement the network function. Following this process is possible to build a *Chain of Trust* (figure 3.10) from physical hardware up to the network service software.

The phase of collecting evidence consists in acquiring all needed measurements and storing them inside the RoTs, which can be a Root of Trust for Storage (RTS) or a virtual RTS (vRTS), then these values will be used for the reporting phase under the assistance of the Root of Trust for Reporting (RTR) or vRTR. The first step in collecting evidence is measuring the load phase that comprehends the boot process until the kernel takes control of the system. Then the OS has to have the components that are responsible for the measurement of applications at run-time. All components necessary during the load time and hypervisor components are measured and stored in the RTS, and it's responsible also to measure the VM's scope and store it in the RTS.

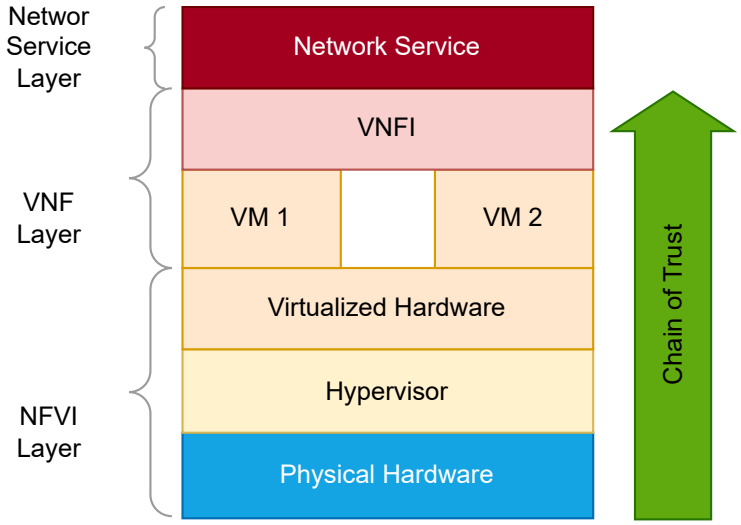

Figure 3.10. Chain of Trust (source: [6])

For what concerns the application sub-scope the measurement process is performed and supervised by the VM's kernel, and when all data has been collected is stored in the virtual RoT, in the vRTS, which is provided by the hypervisor. The vRTS, and generally the vRoT, is used only for what concerns the applications running under the control of the VMs, but not for what concerns the VM platform.

After having acquired and stored all necessary data and measurements for the attestation process, this information will be used to create a report that represents the current state of the system. The generated document has to be sent, in a secure way, to a RAS, which verifies the authenticity of the report, checking the identity of the system. Once this phase terminates with success, the RAS extracts the measurement information from the report and confronts them with some reference values already present in the local environment. If all the measurements match the corresponding reference value or not, the RAS emits the result of the verification. This result can report binary information that determines if the system is trusted or not or can contain a set of information that has to be processed at a later time to come to a final conclusion about the trustworthiness of the system.

Chapter 4

Trust Monitor (TM)

4.1 Overview and Motivation

ETSI defined an entity, called *Trust Manager* [35], inside an NFV infrastructure, more precisely inside the MANO domain, that has the purpose to verify the integrity of VNFs. This entity should be the only one in the infrastructure to attest VNFs, in this way it would be easier to know if a VNF is in the expected state or not because it would be necessary to communicate with the Trust Manager, which can verify the integrity of VNFs in any specific moment. An advantage of this entity is that it can be the only one that implements the logic for the remote attestation, in this way all other entities can be lighter from the implemented logic point of view. A disadvantage of this solution is that the Trust Manager would be a single point of failure, so if this entity would stop working, because of damage or an attack, it would not be possible anymore to verify the integrity of the NVFI. For this reason, it becomes fundamental to protect this system against attacks. ETSI has defined this system it has not developed a common solution for the attestation of network functions.

The *Trust Monitor* (TM), developed by the TORSEC research group of "Politecnico di Torino", is a system proposed to manage the Remote Attestation of VNFs in a SECaaS scenario. It was developed as part of the SHIELD project and presented at the "IEEE Conference on Network Softwarization" [36] in 2019. The TM has been designed to be a stand-alone component in the MANO administrative domain, but not isolated in an NFV platform, which manages the attestation process of vNSFs and NVFI. In addition, being an external component to the MANO entity permits verifying the integrity status of MANO entities as well. It can be considered as an implementation of the Trust Manager proposed by ETSI.

4.2 Architecture

The TM architecture was designed as a modular one, whose schematic representation is represented in figure 4.1, and it provides the following functionalities:

- **Integrity Verification** of both heterogeneous nodes of the NFVI and vNSFs;
- **Notification and Reporting** of integrity status information about the infrastructure to external entities;
- **Audit of Integrity Verification logs** about the infrastructure.

The attestation of the hosts of the infrastructure is performed by the sub-components called *Attestation Drivers* permitting to instantiate different remote attestation workflow, depending on the type of the node, developing them as plugins. This approach permits the TM to manage the attestation of hosts based on different architectures (e.g. ARM, x86), but also with different

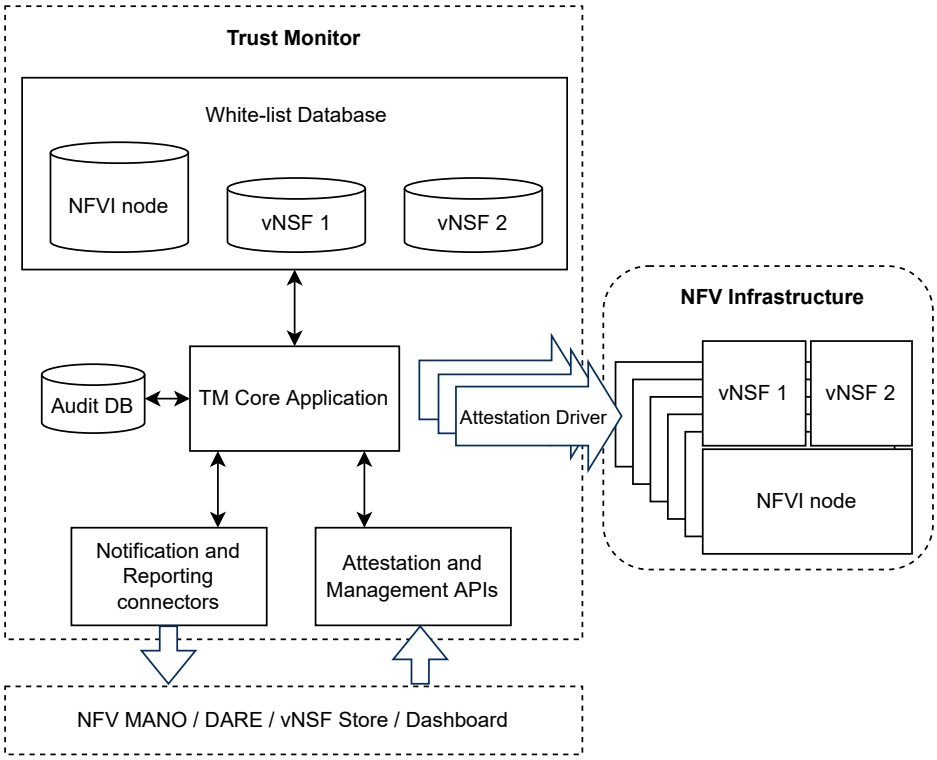

Figure 4.1. Trust Monitor Architecture (source: [36])

RoTs, which could be AMD SEV, Intel SGX, or TPM, in order to not be bonded with just one vendor. This process is available also for *virtulized* TEEs, despite the fact that these offer a lower assurance than hardware-based ones. At the moment, several attestation drivers have been developed to support:

- *Open Attestation* (OAT) framework, which allows the integrity verification of NFVI nodes and vNSFs deployed as Docker [37] containers exploiting the DIVE technology [38], but it supports only version 1.2 of the TPM;
- *Open Cloud Integrity Technology* (OpenCIT), which allows attesting only the nodes of NFVI and it supports both TPM 1.2 and TPM 2.0;
- *Hewlett Packard Enterprise Switch* (HPESwitch), which is used for attestation of SDN switches and controllers;
- *Keylime* framework (this driver has been developed for an updated version of the TM described afterward).

The TM manages the reference values for the attestation of nodes and vNSFs. The databases that store this information are different for NFVI nodes and vNFSs. In the case of NFVI nodes, reference measurements for the Linux distribution installed are used as whitelists. Regarding vNSFs the management of whitelists relies on the vNSF Store, which provides reference measurements for each vNSF provided by the developers or obtained by executing some statistic analysis on the vNSF image. The Whitelist Database is a sensitive component because it stores all reference values that indicate which is the correct state for nodes and vNSFs. For this reason, it is important to use authorization and authentication policies, but also replication and high availability of data are mandatory.

The TM has also an Audit Database that saves attestation results in order that external entities can retrieve historical information about the state of the infrastructure. This is a sensitive

component too, for the data that manages, so it should be protected against attacks like data manipulation.

For the interaction with TM are exposed *Attestation and Management APIs* that permit the registration of NFVI nodes and to start the attestation. Furthermore, via pluggable *Connectors*, are implemented also the *Notification and Reporting* functionalities that permit the interaction with other components defined in a SECaaS scenario:

- *vNSF Store Connector*: it permits to retrieve the reference values for all the vNSFs;
- *Dashboard Connector*: it is the connector that allows the end-user to watch the attestation results;
- *DARE Connector*: The Data Analysis and Remediation Engine (DARE) is the component of an NFV infrastructure that has the purpose to process data deriving from vNSFs to detect possible malicious behaviors. This connector permits to carry of information about the trustworthiness of the infrastructure;
- *vNSFO Connector*: it makes available the possibility to query the vNSF Orchestrator in order to get the list of nodes in the infrastructure and also the list of vNSFs running on them.

4.2.1 Attestation Process

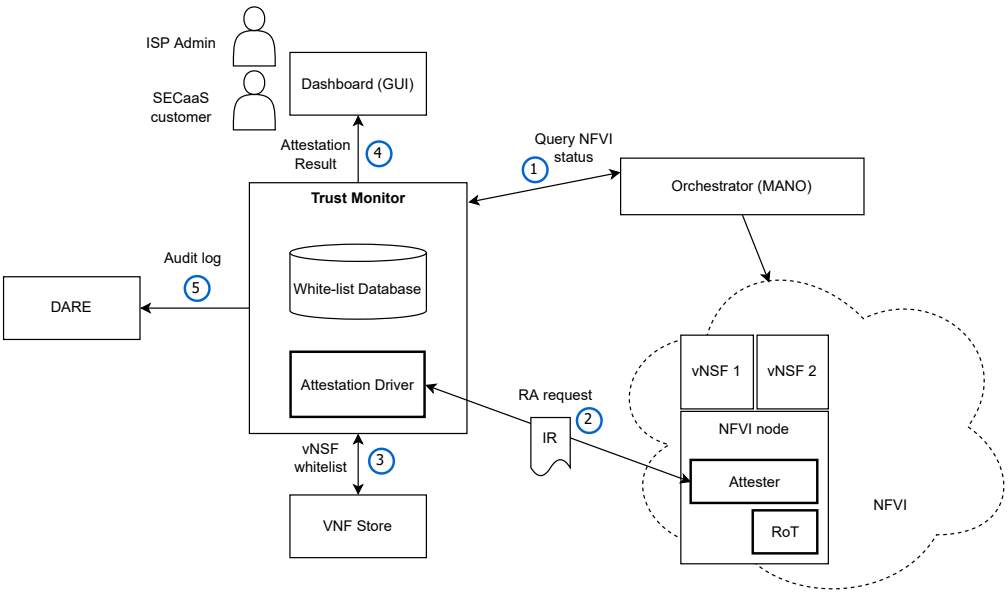

Figure 4.2. Trust Monitor Attestation Process (source: [36])

Figure 4.2 represents the attestation process implemented by the TM and the interactions with components of the infrastructure:

1. The TM retrieves all the infrastructure information, so the list of physical hosts and vNSFs, from the Orchestrator;
2. The TM communicates with the Attester exposed by an NFVI node to start the Attestation Process. The node starts collecting integrity measurements that will use to create an *Integrity Report* (IR) and will send it to the TM;
3. The TM retrieves for each vNSF the list of reference measurements and stores them in the Whitelist Database. Now the TM can verify the authenticity of the IR and confront the values with the reference measurements;

4. The TM produces a notification containing the attestation result, which aggregates information about the trustworthiness of the physical and virtual platforms. This result is forwarded to the Dashboard that provides different views for the ISP Administrator and the SECaaS client;

5. The TM saves in the Audit Database the attestation result and sends it to the DARE which will process it together with vNSFs data.

4.3 Trust Monitor 2.0

As mentioned before, it has been proposed another version of the Trust Monitor with some differences from the original one. This new version was introduced because of the fact that the TM, cause of its modular architecture, can be deployed in other scenarios, in general, cloud deployments based on lightweight virtualization, rather than only the SECaaS one. In fact, the TM 2.0 is able to interact with a container orchestrator (e.g. Kubernetes) providing attestation for applications deployed as sets of containers. The updates in this version are the implementation of a new attestation driver allowing the integration of a custom version of the Keylime attestation framework [39], and the implementation of a Whitelist Web Service that permit to create and manage hosts and containers whitelists. The TM exposes a set of APIs that permits communication with the TM itself in order to control available operations and send all necessary data.

4.3.1 Keylime Attestation Framework

The Keylime framework was proposed by a security research team in MIT's "Lincoln Laboratory". It is an open-source project proposed as a solution for bootstrapping hardware-rooted cryptographic identities for cloud hosts and for periodic attestation of those hosts to verify their integrity in an IaaS environment. Keylime relies on the TPM device functionalities, that permit the management of keys and sensitive data. It was designed to provide several features that are considered needful for a trusted computing system:

- The first feature made available by Keylime is *Secure Bootstrapping* implementing a new bootstrap key derivation protocol that permits injecting identities or other secrets into cloud nodes.

- The second feature provided by Keylime is *System Integrity Monitoring*. This is achieved by performing periodic remote attestation of cloud nodes in order to detect unexpected behaviors and revoke their identity.

- Another feature provided by Keylime is *Secure Layering* which means that the previous functionalities are provided in both bare-metal and VMs.

- The next feature provided is *Compatibility* because it can be integrated with common services in an IaaS scenario (e.g. IPsec, Puppet, Vault, cloud-init).

- Another important feature is *Scalability* because in a cloud context the number of managed nodes can become considerable. For this reason, Keylime is able to manage thousands of nodes and check thousands of IRs per second.

Architecture

The simplified architecture described in this section is referred to the case in which cloud nodes are physical hosts. Components are:

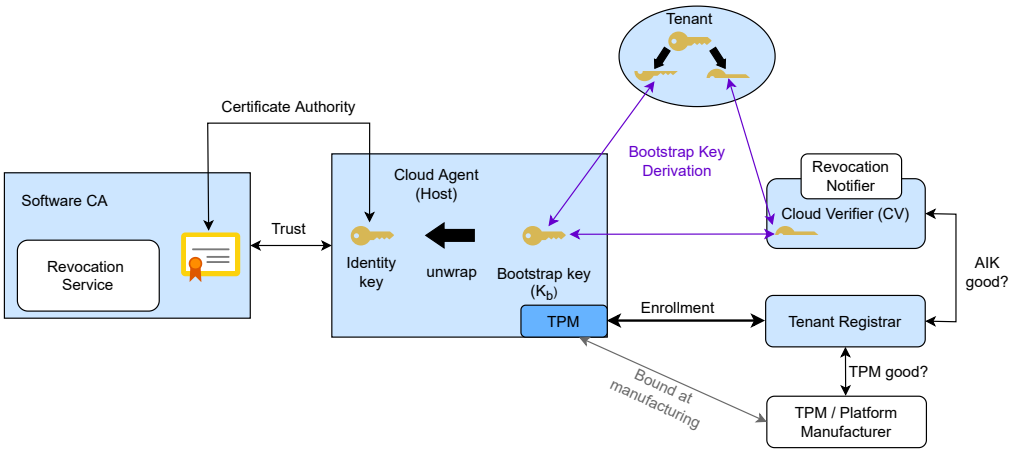

Figure 4.3. Keylime simplified architecture (source: [39])

- **Registrar**: This component stores public part of TPM's keys, AIK_{pub}, *Endorsement Key* (EK_{pub}) and EK certificates indexed by cloud nodes' UUID. It implements the protocol for Attestation Key identity Certification [40], for this reason, it can be considered a simplified implementation of the TCG Attestation CA. Requests for TPM's credentials are managed by the Registrar through an authenticated TLS channel.
- **Cloud Verifier (CV)**: This is the main component of Keylime design. It performs the verification that the tenant's resources are in a trusted state. It retrieves the AIK_{pub} from the Registrar in order to be able to validate a TPM quote.
- **Cloud Agent**: This component runs on each node of the infrastructure and it is responsible to create and send IRs in order to provide information about the current state of the host.
- **Tenant**: This is the component that represents the client of the IaaS resources, in fact, it starts the framework by contacting the agent and sending it some encrypted data related to his service and contacting the CV sending it all the information needed to attest the integrity status of the node that runs the service.
- **Software CA**: This software-only Certification Authority permits binding integrity measurements rooted in the TPM with security services of a higher level.
- **Revocation Service**: This service permits to complete the binding between trusted computing services and higher-level services. This component allows reacting to changes in nodes' states. The procedure begins when a CV detects that a node has become untrusted, so the *Revocation Notifier* contacts the *Software CA* sending it, and to all nodes registered for this service, a "Revocation Event". Once received this message, the *Software CA*, which manages a CRL service, publishes an updated CRL with the information related to the revocation of the software identity key of the untrusted node.

The Keylime attestation framework can be subdivided into four main operational phases:

1. ***Physical Registration Protocol***

   This operation permits the exchange of identity keys and credentials between a host and the framework. When the Cloud Agent starts up it contacts the Registrar, sending all the information related to the identity of the host and the TPM. The Cloud Agent sends to the Registrar its UUID, and the AIK_{pub}, the EK_{pub} and the EK_{cert} of the TPM. Once the Registrar receives all this data, stores it and sends to the Cloud Agent a challenge. If the response of the Cloud Agent is correct, it proves that it posses the AIK_{priv} and the EK_{priv}, so the Registrar can set the Cloud Agent UUID as `active` and starting send its TPM credential when asked.

2. ***Three Party Bootstrap Key Derivation Protocol***

   It can be executed after the node registration protocol, in order to securely deliver the bootstrap key to the new node (after having verified it is in a secure state). This sharing permits verifying the trusted state of the node, so, if the node at the end, is untrusted, it won't receive the second half of the key, and it will not be able to decrypt the payload received. This operation can be divided into three phases. The first one consists of the Tenant that communicates to the Verifier that a particular node exists and sends to the Verifier all the information regarding this particular node (UUID, IP address, port, etc.), together with a part of K_b. During the second and third phases, the CV and the Tenant perform an attestation of the node, at the end of which the node will receive both halves of K_b [39].

3. ***Continuous Remote Attestation*** After the end of the Three Party Bootstrap Key Derivation Protocol, it starts the Continuous Remote Attestation. After having verified that the node is securely booted, so applications will be deployed in a trusted environment, the CV periodically polls the node to monitor its integrity state. This procedure relies on the integrity measurements made by IMA [17] on applications launched on the system. The CV periodically asks for an IR from the Cloud Agent and validates it.

4. ***Revocation Framework***

   When the CV detects that a node is untrusted, it triggers the Revocation Frameworks. The *Revocation Notifier* implements the publish/subscribe pattern, in particular, the CV publishes a new revocation event sending a signed message to the Revocation Notifier, which it sends to all subscribers. A subscriber can be the software CA, the Cloud Agent, and other tools that want to be notified.

4.3.2 Keylime Attestation Driver

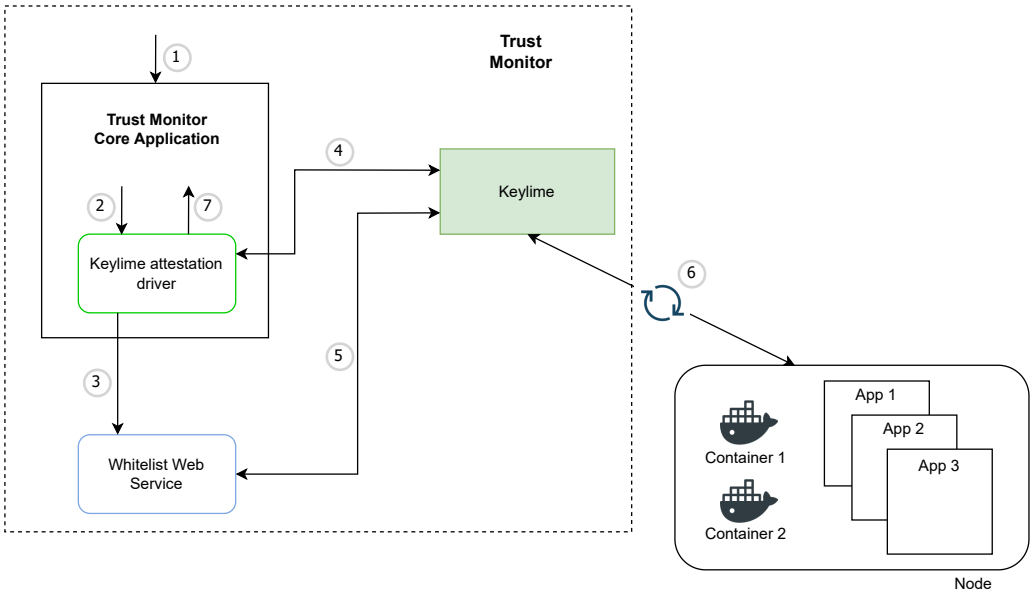

Figure 4.4. Node registration workflow

The Keylime attestation driver is a particular implementation for interfacing with a specific attestation framework. It exposes four methods:

- `registerNode()`: This method permits registering a new node in the attestation framework, which is the operation that enables the attestation process. In the specific case of Keylime, shown in figure 4.4, the workflow starts from (1) the first call to the *register node* API of the

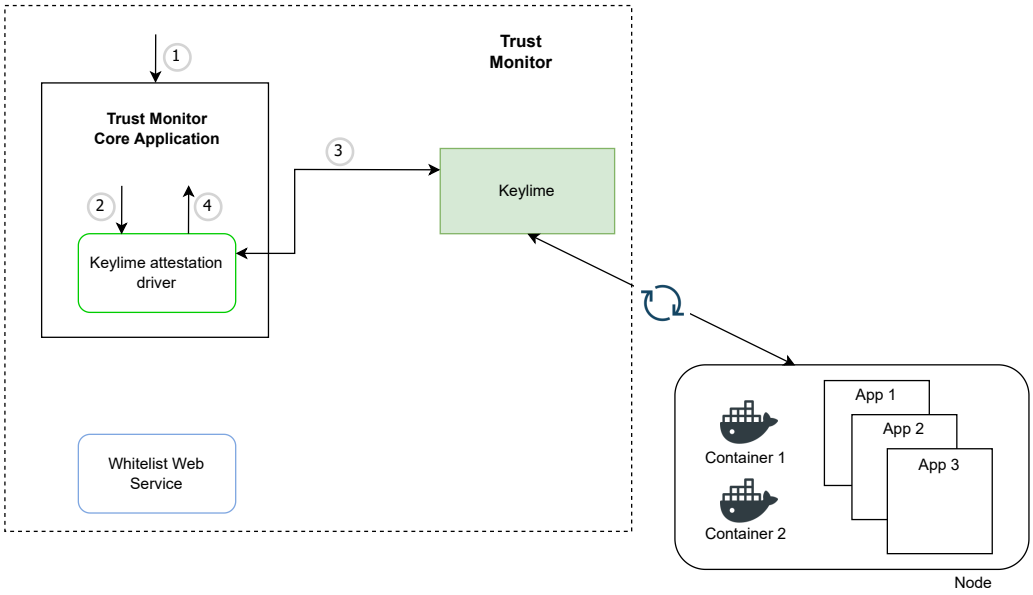

Figure 4.5. Check node integrity status workflow

TM passing all the necessary information about the host (UUID, IP address, OS distribution, list of containers running on the host, etc.). Then the TM calls the registerNode() (2) method of Keylime's driver which asks the Whitelist Service (3) to create the Whitelist correspondent to the host and the whitelist for every container based on the information passed. The driver contacts Keylime (4) passing all the data about the host, passing the URLs from where Keylime can download whitelists for the node and containers. Once Keylime has downloaded whitelists (5) it can register the host and its containers and start the periodic attestation (6). If everything is completed successfully the node is registered in the internal database (7).

- `pollHost()`: This method permits obtaining information about the integrity state of a host. The process workflow is illustrated in figure 4.5. The TM receives a request (1) on its *attest node* API, then it contacts the Keylime driver (2) through the pollHost() methods so the driver will send a request (3) to the Keylime framework in order to obtain information about the integrity status of a host. Once the driver receives all this information it builds an attestation log and sends it back to the TM (4) as a JSON object.

- `getStatus()`: This method is called by the TM to retrieve information about the attestation framework status. Once the TM receives a request on its *status* API it invokes the getStatus() method of the Keylime driver which will contact the framework in order to obtain all the information about its status.

- `deleteNode()`: This method permits to remove a host from the framework previously registered. The workflow of this operation is similar to all others previously explained. The TM receives a request on its *delete node* API that will remove the specified host from the internal database of the TM and subsequently will contact the deleteNode() method of the driver that interacts with the Keylime framework will remove the specified host from the attestation framework.

The structure of a driver is fixed, in this way, the TM can understand and use drivers for several attestation technologies, with the constraint that these drivers respect the expected structure.

4.3.3 Whitelist Web Service

The Whitelist Web Service is a component introduced in the updated TM architecture in order to manage whitelists of hosts and containers in a centralized way. This service has several sources for the reference values, and originally these sources are three:

1. A database that contains the whitelist for the Linux distribution used on all the nodes of the infrastructure;
2. The internal database that saves digests of proprietary software running on hosts in the KnowDigest table;
3. The vNSF Store, which contains whitelists related to containers. It is accessed through the vNSF Store Connector which once retrieved the whitelists stores them in order to use them during attestation.

Having more than one source for whitelists is a cause of increasing the TM core complexity. On the other hand, the concept of a single component that manages whitelists in a centralized way offers a simplified way to access them. In addition, the Whitelist Web Service allows creating a custom whitelist for each node based on its configuration. For this reason, service databases need to be constantly updated, with respect to the software updates uploaded on standard repositories of Linux distributions.

4.4 Criticalities and open challenges of the classic version

The Trust Monitor introduces several advantages for managing the continuous remote attestation of infrastructure because it adds a level of abstraction to the attestation framework allowing to use more than one. This permits a more flexible utilization of these technologies. The goal of this project is to achieve the possibility to manage attestation in a more heterogeneous environment which could be an NFV infrastructure but also a more general purpose cloud environment that allows the user to instantiate different objects, such as VMs, containers, also pods (in a Kubernetes context), enclaves (in a TEE context) and more. For this reason, a very important improvement is implementing the possibility to manage the most generic objects, in order to achieve the wanted flexibility.

Another improvement that is necessary is making the TM completely independent from attestation drivers because in the original version attestation drivers are developed as plugins, but there is still a bond between attestation drivers and the TM core application. An example could be the Keylime attestation driver that implements the interface with Keylime, but in the `pollHost()` method implements a single request for the integrity status of the node. The logic for continuous attestation is implemented in the TM core application, so the TM needs to be aware of the attestation technology used. The goal should be to move the whole attestation process to the attestation driver, in this way the TM could use them without knowing anything about the internal implementation.

There are other possible updates needed, for example implementing a process to build whitelists in an automatic way permitting considerably increasing the flexibility of the entire system. This improvement would permit to deploy nodes (physical or virtual), applications, services, and more not providing a fixed whitelist for each one, but deriving them from some analysis of those systems in order to achieve a more automatic way to deploy them in a secure way having the possibility to continuous attest them.

Chapter 5

Trust Monitor Redesign

This chapter focus of illustrate the new architecture of the Trust Monitor designed during the development of this thesis work. It will consider several aspects of the components and the purpose of each one and how they collaborate together.

5.1 Architecture of new Trust Monitor

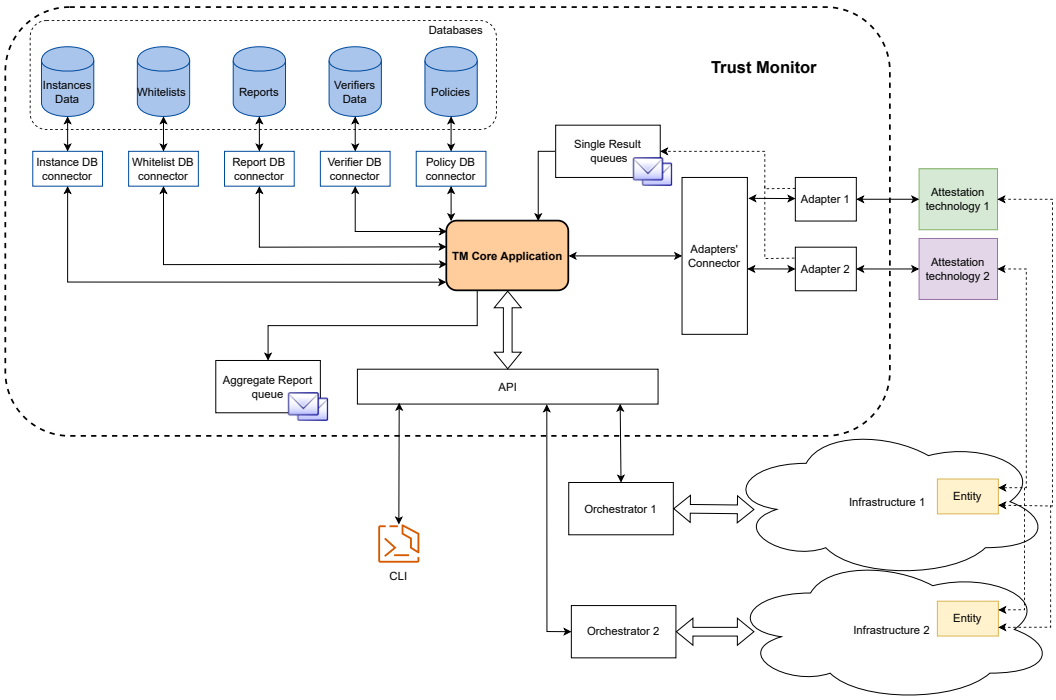

Figure 5.1. New Trust Monitor Architecture

The architecture of the new Trust Monitor is based on the classic implementation, in order to maintain the same basic structure. The new concept is making the TM more independent from the attestation technologies used and the kind of infrastructure. The new architecture is shown in figure 5.1 and represents all components that are part of it.

The key concept of this new version is making the Trust Monitor more independent from the attestation technologies used to verify the integrity of the objects on which it is intended to

perform the remote attestation. This goal is obtained by moving the most significant logic for the attestation on *Attestation Adapters*, which are very similar to the Attestation Drivers of the previous version of the TM. This idea permits to maintain the TM unaware of which attestation technologies are used, so it has the purpose to contact the correct Adapters, specified for each entity registered, and then managing their results. Once received, these attestation results are aggregated into a report for each entity that has an active remote attestation process. The report produced permits to have an instant view of the status of the object under evaluation. This process is possible thanks to message queues that permit sorting attestation results coming from different attestation technologies in a quick and easy way. The TM core application in this way can obtain this information in an asynchronous way that allows elaborating other requests during the wait. Some queues receive attestation results from all the attestation adapters and permit the TM core application to read those results and aggregate them based on the entity. Another queue, on the basic logic proposed, receives all the reports created by the TM core application in order that a custom consumer, developed ad-hoc based on the necessities, can process those reports and make some decisions.

The TM exposes APIs that permit controlling and managing it. These APIs are the interface with the administrator, they allow to manage all other components and the attestation process.

Other fundamental components are the databases, that store all the information needed for the attestation process. In this way, it is possible to maintain a different state from the attestation frameworks used. This permits to store information independent of the attestation process and independent of the attestation frameworks that will be used.

5.2 Level of Abstraction

The purpose of the TM is to add a level of abstraction to the remote attestation process. Adding this layer permits obtaining an independent workflow for remote attestation. Once all information is stored in the TM, such as data about objects to attest, data related to verifiers and attestation technologies that will be used, the entire process of attestation becomes independent of the technologies used because the only system contacted to start the integrity verification is the TM which then will interact with all required attestation technologies. This approach allows attestation of an entity with more than one technology because it will be only necessary to specify which technologies will be used for remote attestation on that particular object and then when the process will start, the TM will manage all technologies specified and will aggregate all results in a single report.

5.2.1 Attestation Adapters

The possibility to manage the attestation process in a more flexible way is obtained through the ***Attestation Adapter*** which is the key to the abstraction brought by the TM. This component permits to connect an attestation framework with the TM core application but without giving to the TM itself any information about the framework used. The idea is very similar to the previous versions of the TM, in fact, an adapter must have a fixed structure in order to expose to the TM core application a common interface allowing to use it without knowing anything about its implementation and the framework attestation workflow. Each adapter can be developed following the best approach for the specific attestation framework to which is referred but following a specific schema in order that the TM can understand it.

The flexibility of this system is also thanks to several *metadata* fields present in the databases. These fields permit to save custom data that will be obscure to the TM core application but can be used by the adapters, so developing a custom adapter is possible to design some custom data to store in databases that will be useful during the remote attestation phase. In addition, the structure of whitelists is also customizable because different attestation technologies can support different whitelist structures, so also in this case the structure of whitelists will be obscure to the TM core application and will be clear to the specific attestation adapter which will be able to manage it and use it for the attestation process.

5.3 Description of the Componets

5.3.1 TM Core Application

The TM Core Application is the main component that manages all the requests about the attestation processes. It manages the central high-level logic, so it does not manage directly the attestation process of an entity, but when it receives a request that specifies an entity on which start the remote attestation, it retrieves all necessary data, contacting the *Database Connectors*, and pass it to the adapters that will manage the remote attestation process. The TM Core Application then receives all the attestation results sent by the adapters. In addition, the TM Core maintains the memory of attestation processes that are currently running making available the possibility to retrieve externally this state in order to be able to know the actual state of the TM at any moment.

The logic of the TM Core is quite simple because, as already reported before, the purpose of this work is to move the attestation logic on the adapters in order to maintain the TM Core more light and flexible.

5.3.2 Connectors

The Connectors (figure 5.2) are the interface of the TM Core with all other components of the system. They permit to send and receive data between the Core Application and other elements. The connectors that allow this flow of data are the *Database Connectors*, which permit the databases to be queried by the TM Core in order to retrieve or store data. These are modules which the purpose to implement the necessary logic to contact the database, which uses different technologies. In this way, the technology used for databases is not important for the TM Core Application because it will contact the connector without knowing anything about the database implementation.

The *Adapter's Connector* has the more important role to interface the TM Core Application with all the attestation adapters deployed. The purpose of this module is to contact all the adapters needed for the attestation of each entity. This is possible thanks to the *dynamic loading* of the adapters, which permits loading all the adapters that are declared in a configuration file. This mechanism increases considerably the flexibility of the system because to add or remove an attestation technology is only necessary to edit a configuration file in order to make load a specific adapter or not.

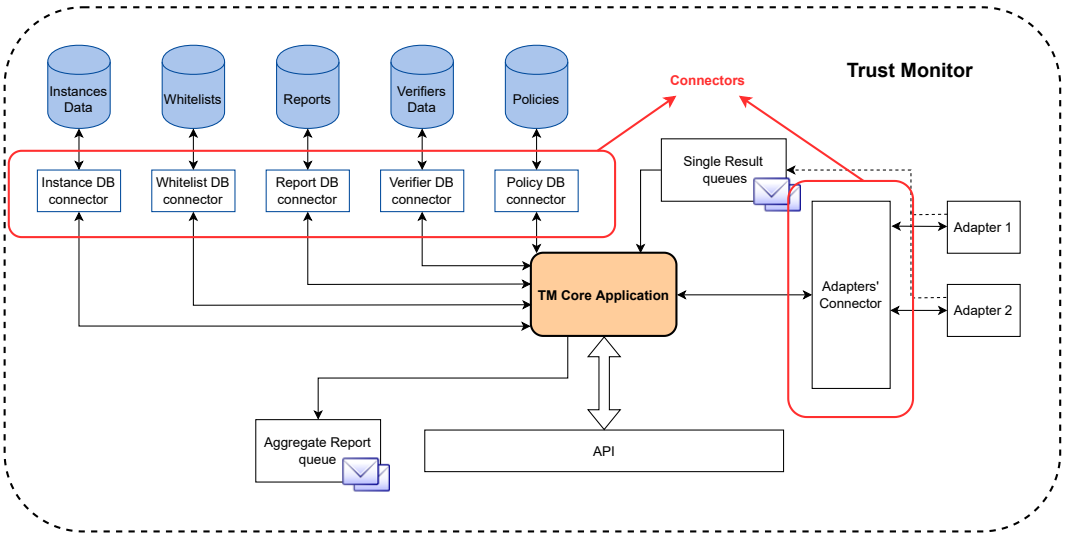

Figure 5.2. Trust Monitor Connectors

5.3.3 Databases

The first components that have been redesigned are databases in order to satisfy the requirements necessary for each one. These are crucial components because they store information that permits the verification of the integrity of infrastructure's objects and allows to save those results in order to be able to perform analysis with them. For each database is available a *Connector* that permits the TM core application to interface and communicate with them. The total number of databases is five and they are:

- Instances database
- Whitelist database
- Report database
- Verifier database
- Policy database

Instances Database

The Instances Database is the main database that contains all the information related to each object that will be attested. It is a relational database has one table called *entities* and the schema contains the following attributes:

- `entity_uuid`: This is the primary key of the table and it is an internal identifier for the single object to attest. It is assigned by the outside at the moment of the registration of the entity and it will be used for all the operations exposed by the TM on entities.
- `inf_id`: This attribute allows to identify the infrastructure to which the entity belongs.
- `att_tech`: This attribute is a list of attestation technologies that will be used to verify the integrity of the specific entity. This permits to be able to use more than one attestation technology for each object.
- `name`: This field represents the name of the object. It is assigned at the moment of registration and it has no impact on the logic of the TM, but it can be useful for quicker identification of entities.
- `external_id`: It is an identifier of the entity external to the TM. It is assigned at registration time by the outside and it permits to define an identifier that can be used for example by an attestation technology.
- `type`: This attribute represents the type of the entity, such as node, VM, container, etc.
- `whitelist_uuid`: This value is an external reference to the whitelist database, in order to link an entity to a whitelist, which will be used during the attestation process.
- `child`: This attribute is a list of `entity_uuid` values, which permits to know the objects contained in another one. For example, a physical node can have a list of containers running on it.
- `parent`: This value has the opposite meaning of the previous one. In this case, it represents the `entity_uuid` of the entity that contains the represented object.
- `state`: This value represents the state of the entity in the TM in order to be able to understand which process is running related to the specific entity.
- `metadata`: This is an important field because it represents in some way the flexibility of the TM. Inside this field can be stored custom information, that the TM interprets as a *blob*, so this data is not relevant for the primary logic of the TM, but they can be used by attestation technologies, which could need some additional information to properly work.

Whitelist Database

The whitelist database store all information about reference values for the remote attestation. These values are not used directly by the TM, because it has the purpose to manages different attestation technologies and aggregating their attestation results, so these reference values will be retrieved from the whitelist database and passed to the respective attestation technology. In particular, it is a NoSQL (non-relational) database that allows storing whitelist with different structures depending on the attestation technology used. The document associated with a specific whitelist contains two fields: the first one is dedicated to `metadata` in order to be more flexible and be able to define some custom values; the second one, `whitelist`, is dedicated to containing the reference values in the most suitable structure.

Report Database

The Report Database permits to store reports produced by the TM during the process of aggregation of attestation results received from attestation frameworks. The presence of this component allows doing historical analysis of attestation results related to specific entities. The document that stores a report contains the time of creation, the state of the entity at that particular moment (based on some policies) and the list of attestation results for every attestation technology that is performing integrity verification on that particular entity.

Verifier Database

The Verifier Database has the purpose to memorize the information related to the attestation technology of each infrastructure managed by the system. In this case, there is only one field associated with an attestation technology which is a `metadata` field. The presence of this attribute permits to store custom information, in a structured way, about a Verifier that will be used during the remote attestation phase. The metadata attribute is treated the same as the metadata attribute of the Instances Database, so it is not interpreted by the TM core application, which manages it as a *black box* but will be understood by the proper adapter, that will use that information to interact with the specific attestation framework.

Policy Database

The Policy Database allows storing policies, bound with entities, which will be used during the aggregation phase, so during the construction of the report. In this way, it is possible to have different results about the trustworthiness of an entity depending on the policies specified.

5.3.4 Queues

In the TM architecture are present two kinds of queues, the *Single Result Queue* and the *Aggregate Report Queue*. The first one has the purpose to collect all the attestation results coming from the attestation technologies through the adapters. This module permits the TM Core Application to receive attestation results in an asynchronous way allowing a more optimized managing of them in order to create reports that aggregate these results following the policies specified for entities. In this case, there will be a single queue for each entity under integrity verification. The second one has the purpose to make available for live consummation the report created. In this way, it is possible to develop a custom consumer that can elaborate on reports produced a make some actions or decisions based on them. In this case there will a single queue.

5.4 Interfaces and High-Level Workflow

The interfaces that the TM has to relate to other external systems are the Adapters, which permit communicating with attestation technologies, and the *APIs Manager*. The APIs Manager is the

component that has the purpose to control and serve all the requests coming from an external system. It makes available several possible operations that permit to correctly perform the remote attestation process.

5.4.1 TM Operations

All the operations made available by the TM are managed by the APIs Manager which receives the requests and then contacts the TM Core Application which will perform the procedure requested. The operations permitted are:

- **Register Entity**: This operation permits to store a new entity in the Instances Database, in this way is possible to specify all the information needed for the attestation of that specific object. In this case, data are only stored in the TM database without contacting any attestation adapters.
- **Edit Entity**: This operation makes available the possibility to edit one or more attributes of a specific entity stored in the TM database. This operation modifies only the information stored in the Instances Database without contacting any adapter.
- **Delete Entity**: Requesting this operation is possible to delete an entity from the Instances Database, but also in this case the information modified is only in the TM database, so no adapter is contacted.
- **Add Whitelist**: This operation, similar to the Register Entity, permits to save a new whitelist in the Whitelist Database of the TM. This whitelist will be linked to a specific entity during the registration of the entity or during the edit operation.
- **Delete Whitelist**: This operation permits to delete a whitelist from the TM database. Also in this case is an action limited to the TM context.
- **Add Verifier**: This operation permits to add all the information needed by a specific attestation framework related to a specific infrastructure. This operation doesn't give any information about the adapter to the TM, it permits only to store the necessary information to be able to contact the framework.
- **Delete Verifier**: This operation permits to delete the information about a particular attestation technology. The adapter will still be loaded because this action modifies only the information stored in the Verifier Database.
- **Add Policy**: This operation permits to add a new policy for a specific entity that will be used during the creation of the report once received all the results from the attestation technologies that are performing the remote attestation of that particular entity.
- **Delete Policy**: This operation removes a policy for a specific entity from the Policy Database.
- **Start Attestation**: This is the main operation that permits, once all the necessary data are stored in the TM databases, to start the attestation of a specific entity. This procedure, as shown in figure 5.3, begins with a request to the APIs manager (1) that contact the TM Core Application which retrieves all the necessary data (2) (3) (4) to start the remote attestation process. Once the TM Core has retrieved all the information it can contact the Adapters' Connector (5) which selects the correct adapter. In case it is used more than one attestation technology the Adapters' Connector will contact all the necessary adapters. Once contacted, the selected adapter starts the communication with the attestation framework (6) in order to control the attestation process performed by the attestation technology (7). The results of the process are published on the Single Report queue (8) which makes them available to the TM Core Application, which read them (9) and periodically produces a report that aggregates all this information publishing it on the Aggregate Report queue (10). Reports are also stored in the Report Database which will permit to perform historical analysis on all reports created for a specific entity.

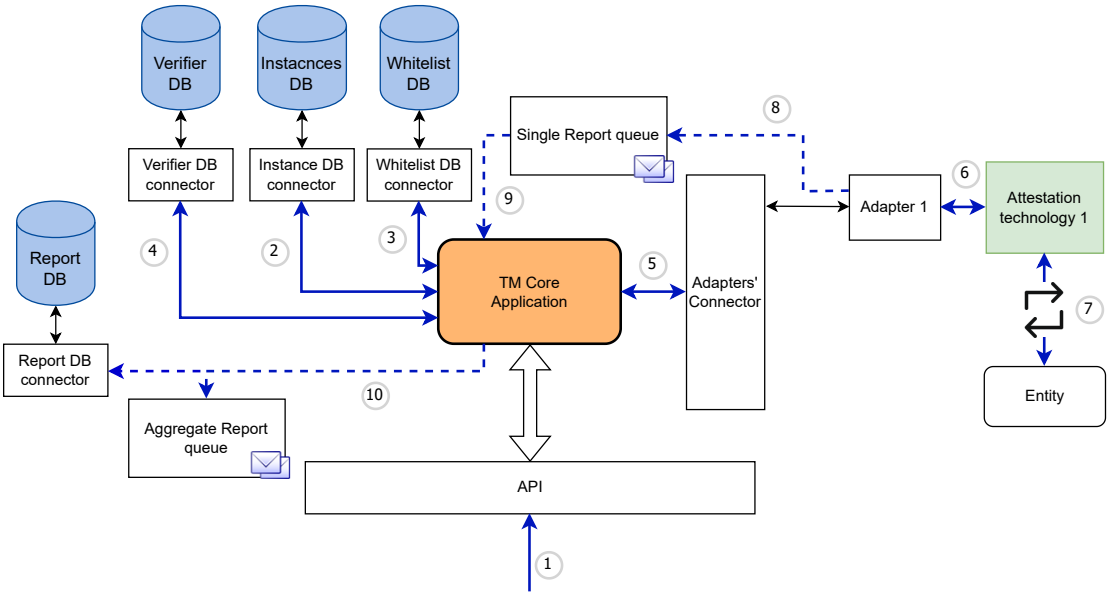

Figure 5.3. High-level workflow for entity attestation

- **Stop Attestation**: This operation permits stopping the attestation process on the attestation framework and on the TM.

Chapter 6

Trust Monitor Implementation

This chapter explains the new implementation of the Trust Monitor component by component, the tools and libraries used and the motivation behind the choice of these technologies. As already described in the previous chapter, the Trust Monitor is composed of several components. Each component has been developed as a file that exposes some methods in order to interface with other components. The entire Trust Monitor system has been implemented using the Python [41] programming language. The structure has several files as shown below:

```
trust-monitor
├── adapters
│   └── keylime.py
├── databases_connectors
│   ├── instances.py
│   ├── reports.py
│   ├── verfiers.py
│   ├── whitelists.py
│   └── policies.py
├── kafka_connectors
│   ├── kafka_connector.py
│   └── report_reader.py
├── docker-compose
│   ├── docker-compose.yml
│   └── init-docker.sh
├── adapters_connector.py
├── api-manager.py
├── core.py
├── config.ini
└── Dockerfile
```

Some of these files are the actual implementation of the logic and some are configuration files needed for the execution or deployment.

The entire system is deployed as a set of Docker containers using the `docker-compose` tool. This tool permits to set up and run several containers at the same time, specifying all the configurations in a YAML file (`docker-compose.yml`). The set of containers, shown in figure 6.1, is composed of four containers: one for the NoSQL databases, one for the relational databases, one for the queues and the last one for the TM core logic. The `init-docker.sh` file is a Bash script that initializes databases after the first deployment of them.

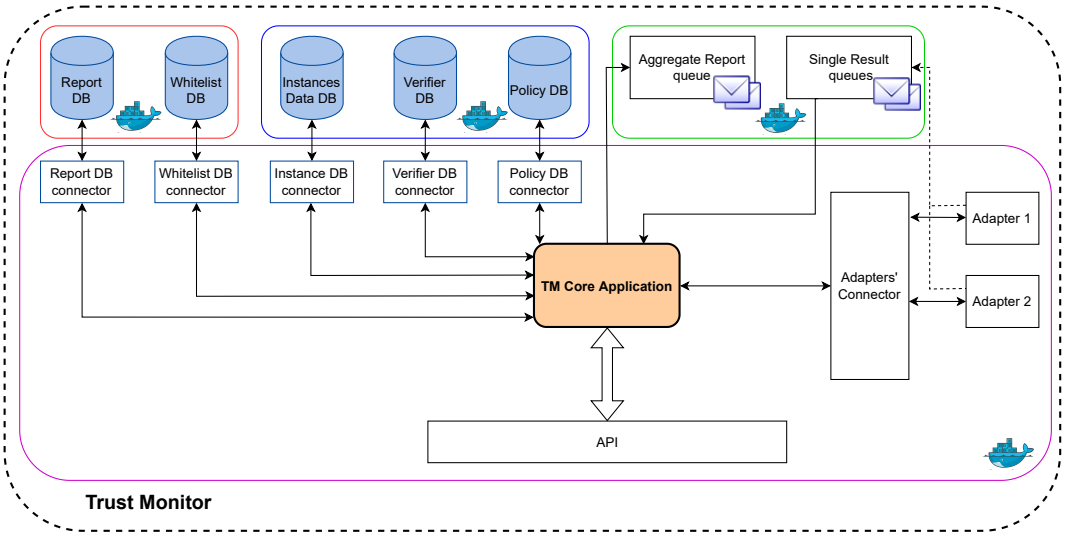

Figure 6.1. Trust Monitor containers deployment

6.1 Tools and Libraries

The TM system, as already reported before, has been developed using the Python programming language, version 3.8, in order to maintain compatibility with previous versions of the TM. Each logical component has been developed as a single python file which exposes all the necessary methods in order that other components can interface with it.

6.1.1 Configuration File

All the configurable information is saved into a configuration file called `config.ini`. This file is an INI file (an example is shown in listing 6.1), which has a specific syntax and structure which is based on the *key-value* structure. All necessary values are represented in this form and in addition, there is also the possibility to divide these values in *sections*, in order to have a more understandable structure.

```
[section_a]
key_a = value_a
key_b = value_b

[section_b]
key_c = value_c
key_d = value_d
```

Listing 6.1. INI file example

This file permits to specify all configurable parameters, for example, it is possible to declare the adapters to load at run time, specifying the name of the adapters in the section `adapters`.

In order to load this file during the TM execution, it has been used the `configparser` python library, which permits to dynamically load the configuration file and saving all the information into a python dictionary in order to make them available during the execution. An example is reported in listing 6.2 that is the result taking as configuration file the example reported in listing 6.1.

```
{
    "section_a": {
```

```
        "key_a": "value_a",
        "key_b": "value_b"
    },
    "section_b": {
        "key_c": "value_c",
        "key_d": "value_d"
    }
}
```

Listing 6.2. `configparser` dictionary example

6.1.2 APIs Manager

The APIs Manager component is implemented in the `api-manager.py` file which is the first file launched in order to start the TM execution. This file contains the definition of all the logic in order to serve the various operation permitted. To implement this component, which behaves as a web server, it has been used the *Quart* python framework. Quart is an *asyncio* reimplementation of the popular Flask microframework API, this permits to manage requests in an asynchronous way without blocking the flow of execution until the request is served. Asyncio is a component of the Python standard library and it provides an event loop with input/output operations, so this permits to implement concurrency and obtaining better performance in CPU utilization. In listing 6.3 is shown how this logic can be implemented in order to obtain the concurrency execution of different tasks that can terminate in different moments but each task is executed in parallel with others in order to not block the flow of execution.

```
import asyncio


async def simulated_fetch(url, delay):
    await asyncio.sleep(delay)
    print(f"Fetched {url} after {delay}")
    return f"<html>{url}"


def main():
    loop = asyncio.get_event_loop()
    results = loop.run_until_complete(asyncio.gather(
        simulated_fetch('http://google.com', 2),
        simulated_fetch('http://bbc.co.uk', 1),
    ))
    print(results)
```

Listing 6.3. Asyncio demonstration (source: [42])

This behavior is very suitable for a web server because it receives asynchronous requests and in this way is possible to serve them in parallel independent of their complexity. As already mentioned, Quart is a reimplementation of Flask so the syntax remains very simple and understandable as shown in listing 6.4.

```
from quart import Quart

app = Quart(__name__)

@app.route('/')
async def helloWorld():
    return 'Hello World!'

```

```
app.run()
```

Listing 6.4. Quart example

6.1.3 Databases

The databases are realized with two technologies: PostgreSQL for relational databases, which are the Instances database, the Verifiers database and the Policy database, and MongoDB for NoSQL databases which are the Whitelists database and the Report database.

In the case of relational databases, the choice of PostgreSQL is motivated by the need for some native way to can manage short information and JSON objects. This technology, in fact, permits to store JSON objects in relational tables with native management of them. For this reason, PostgreSQL makes available the `json` type in order to directly manage JSON objects without the need to convert them to `string` or other types. It supports SQL so all the operations can be done using it.

In the case of NoSQL databases, MongoDB was chosen because of its performance, because the documents managed, for example in the Whitelists database, could become very big, so there was the necessity for a system that can manage very big documents. In this case, with the fact that MongoDB is a NoSQL database, it doesn't support SQL, so all the operations are managed by a specific query language. In addition, also the structure is different from a relation database, in fact, there are no tables but *collections* which substitute them. A collection can contain several *documents* (an example is shown in listing 6.5) that are the equivalent of an object saved in the database which is the equivalent of a tuple in the case of a relational database. The structure of a document is that of a JSON object where there can be several attributes, and each attribute can be another JSON object, an array or just a value. The only constraint is that a document must contain the `_id` attribute which is the identifier of that particular document. It can be set manually when the object is inserted into the collection, in any case, if this attribute is not passed when the document is inserted, it will be generated automatically.

```
{
    "_id": 1,
    "first_name": "Mario",
    "email": "mario@polito.it",
    "cell": "765-555-5555",
    "likes": [
        "fashion",
        "sports",
        "shopping"
    ],
    "businesses": [
        {
            "name": "Entertainment 1080",
            "partner": "Rosa",
            "status": "Bankrupt",
            "date_founded": {
                "$date": "2012-05-19T04:00:00Z"
            }
        },
        {
            "name": "Best of Sports",
            "date_founded": {
                "$date": "2012-11-01T04:00:00Z"
            }
        }
    ]
```

```
}
```

Listing 6.5. MongoDB document example

6.1.4 Queues

To implement the necessary queues it is used Apache Kafka [43], which permits to implement a *publish/subscribe* protocol used to transmit attestation results and aggregate reports. In addition, it is used this technology to maintain compatibility with previous versions of the TM. Messages sent in this infrastructure are organized in *topics*, which can be considered similar to folders of a filesystem. Each topic can have more than one producer, that publishes messages on the topic, and more than one consumer, that reads and processes messages sent on the topic. In the particular case of the TM (figure 6.2) the topics are of two kinds:

- `result_entity_{entity_uuid}`: for these topics producers are all the attestation technologies that are running. Each attestation process in execution publishes its result on a specific topic created when the attestation process starts. The consumer is the TM itself which reads these results and aggregates them into reports for each entity under verification.
- `report`: the producer for this topic is the TM which after having produced reports that aggregate the attestation results received, it publishes them on this topic in order to make them available for processing. The consumers for this topic are not defined, because the TM makes available reports on this topic but the processing and the decision taken from these reports must be implemented by the user that decides how to use these reports.

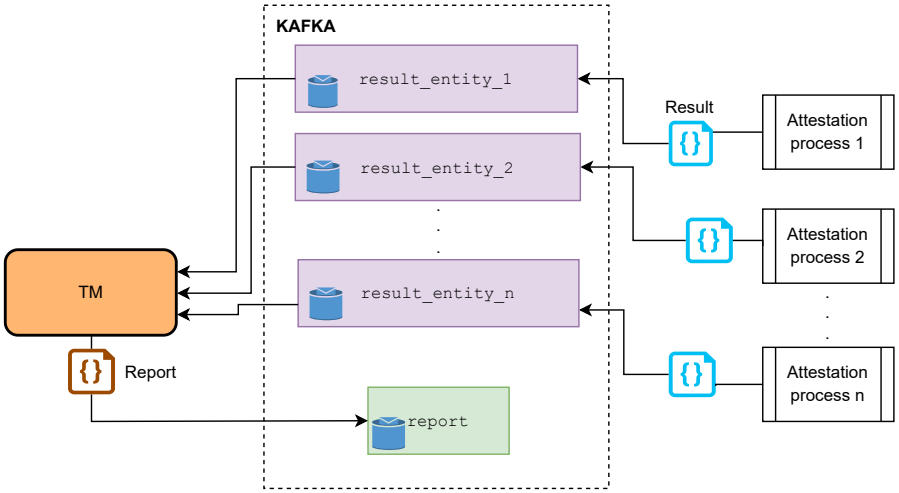

Figure 6.2. Results and reports workflow

6.1.5 Databases' connectors

Databases' connectors implement all the functionalities needed in order to store and manipulated data that are necessary for the remote attestation processes. Each connector is implemented as a python. Each connector validates objects received from requests, and this validation is performed using the `jsonschema` library which is an implementation of the JSON Schema specification [44] for python. The concept is specifying a base schema of the expected JSON object in order to check if the received object has the required characteristics or not. The schema is declared as a JSON object (listing 6.6) which specifies the attributes required and the type of each attribute and if the data of the request doesn't match this schema is discarded.

```
schema = {
    "type" : "object",
    "properties" : {
        "name" : {"type" : "string"},
        "price" : {"type" : "number"}
    },
}

{
    "name" : "Eggs",
    "price" : 34.99
}                       # Valid object

{
    "name" : "Eggs",
    "price" : "Invalid"
}                       # Invalid object
```

Listing 6.6. JSON Schema example

Two different libraries are used in order to create database connections. The first one is `psycopg2` that permits to manage PostgreSQL databases. The second one is `pymongo` that permits to manage MongoDB databases.

6.2 Low-level workflow of the solution

6.2.1 Adapters

An Adapter is the main component for the interface with the specific remote attestation technology, in fact, it is the element that implements the logic for managing the attestation process directly on the attestation framework. In order to obtain the feature of dynamic loading of adapters has been defined a fixed structure (listing 6.7) for each adapter in order that the methods exposed are always the same.

```
class className:

    def register(entity, whitelist, verifier):
        ...

    def attest(entity, verifier, whitelist, se, topic):
        ...

    def delete(entity, verifier):
        ...

    def status(verifier):
        ...
```

Listing 6.7. Adapter's structure

The structure of the file must contain a class that must implement three methods:

1. `register(entity, whitelist, verifier)`: This method receives all information about the entity to attest, about the whitelist and the verifier to contact.
2. `attest(entity, verifier, whitelist, se, topic)`: This method receives all information about the entity to attest, the whitelist and the verifier to contact. In addition, it

receives `se` which is an `Event` object of the standard python library `threading` that permits set a termination condition for a thread. This object can be useful if the adapter needs to implement a loop for the attestation process and it is performed by a secondary thread. The last parameter is the topic on which publish the attestation results received from the attestation framework.

3. `delete(entity, verifier)`: This method receives all information about the entity and the verifier to contact.
4. `status(verifier)`: This method receives only the information about the verifier in order to be able to retrieve its status.

Each adapter developed needs to be declared into the configuration file in order to be loaded by the TM. This declaration has to be done in the `[adapters]` section of the `config.ini` file, inserting a line with the following structure:

```
<name_of_the_adapter_file> = <name_of_the_class_implemented>
```

In addition, an adapter in order to be loaded needs to be placed in the `adapters` directory of the project.

Keylime Adapter

In order to be able to test the system has been developed an adapter for the Keylime framework. In this case, the version of Keylime is a custom one, to which has been added the feature of pods attestation in a Kubernetes infrastructure.

In this case, the whole attestation logic is implemented in the `attest` method, because in the case of Keylime the registration and the delete of an object are done respectively at the start of the attestation and at the end of the attestation. The implementation of the `attest` method of the Keylime adapter is based on a loop. Before the loop, its is performed the registration of the object into Keylime. In the case the entity to attest has the `child` field set, it retrieves all the information about the pods that are running on that particular entity. After having retrieved all information it sends a request on the registration API of the Tenant which will perform it and start the remote attestation. Inside the loop, it is performed a cyclic request to the Tenant asking for the attestation result related to the entity under verification. Once received a result it publishes it on the `result_entity_{entity_uuid}` topic which will be processed by the Kafka consumer that once received a result for each attestation technology that is performing remote attestation on the entity, it produces a report and publishes it on the `report` topic. When `se` is set the loop stops and the entity is deleted from Keylime which is the operation that stops the remote attestation on it. The `attest` method is executed by a dedicated thread (figure 6.3) which permits it to be executed in parallel with other tasks.

6.2.2 TM Core Application

The TM Core Application is implemented as a set of methods called by the APIs Manager in order to interface with all other components of the infrastructure. In order to maintain the attestation status, the TM store the information about each attestation process in a dictionary, which saves data about the Verify thread and the corresponding stop event`se`. In this way, when the DELETE `attest_entity` is called for an entity, the TM can locate the specific thread and terminate it in order to stop the remote attestation process.

The attestation process, shown in figure 6.4, starts at the moment the APIs manager receives a request on the POST `attest_entity` API. Once received this request, the TM core retrieves all the information about the entity, the whitelist and the verifier, then launches a thread (Verify thread) that will manage the entire process and creates the specific topic, in this way it can serve several requests in parallel. The Verify thread launches two other threads. The first one is the Kafka consumer thread that will perform the consumer which reads attestation results on the `result_entity_{entity_uuid}` topic and produces the reports. This thread is linked to

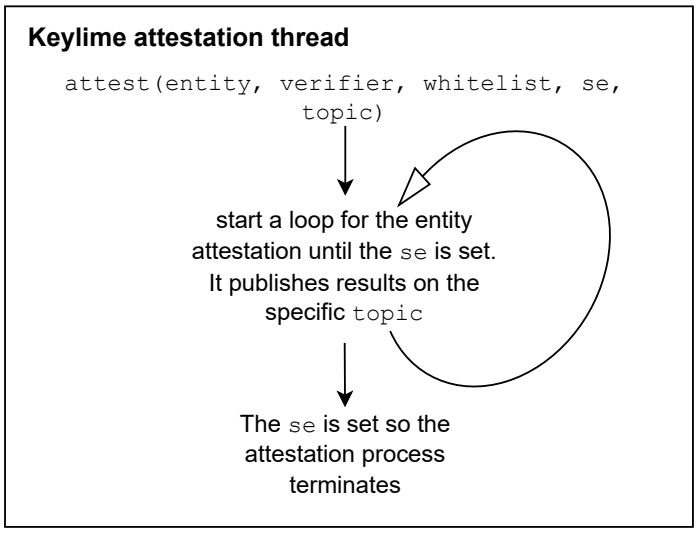

Figure 6.3. Keylime attestation thread

a stop event (`stop_event`) which will be set at the moment when the attestation process will terminate, in this way the thread will terminate. The second one is the Attestation thread which executes the `attest` method of the adapter, in this particular case, the Keylime adapter. This thread is linked to a stop event (`se`) too, which will be set when it's received a request on the DELETE `attest_entity` API. The Verify thread, once launched the Kafka consumer thread and the Attestation thread wait for the Attestation thread to terminate. When the Main thread set `se` the Attestation thread stops the attestation on the attestation framework and terminates. Once it is terminated the Verify thread set `stop_event` and the Kafka consumer thread terminates too. At this point, the Verify thread joins the Kafka consumer thread and terminates.

6.2.3 Adapters' connector

The Adapters' Connector is the component that permits the TM to interface with different remote attestation technologies without being aware of which one is contacting, in order to be able to add and remove attestation technologies without having to modify the TM core. In order to do so, this component uses the `[adapters]` section of the configuration file. The key of each pair identifies the file where it can be found the module to import. After having collected all this information and imported the declared adapters, it is performed a check when a method is invoked in order to verify that an adapter has implemented that specific method.

6.3 APIs and Operations

Each TM operation has a corresponding API in order to be invoked. The available APIs are:

- `/entity`: It permits registering, retrieving, deleting or modifying an entity in the Instances database. The registration needs to be called with POST method, the updating with PUT method and the deletion with DELETE method. To retrieve information about an entity the API needs to be called with GET and the URL must include the `entity_uuid` in the form `/entity?entity_uuid=<id>`.

- `/attest_entity`: This API called with method POST permits to start the periodic remote attestation of an entity already registered in the TM. When it is called with method DELETE permits to stop periodic remote attestation for the specified entity.

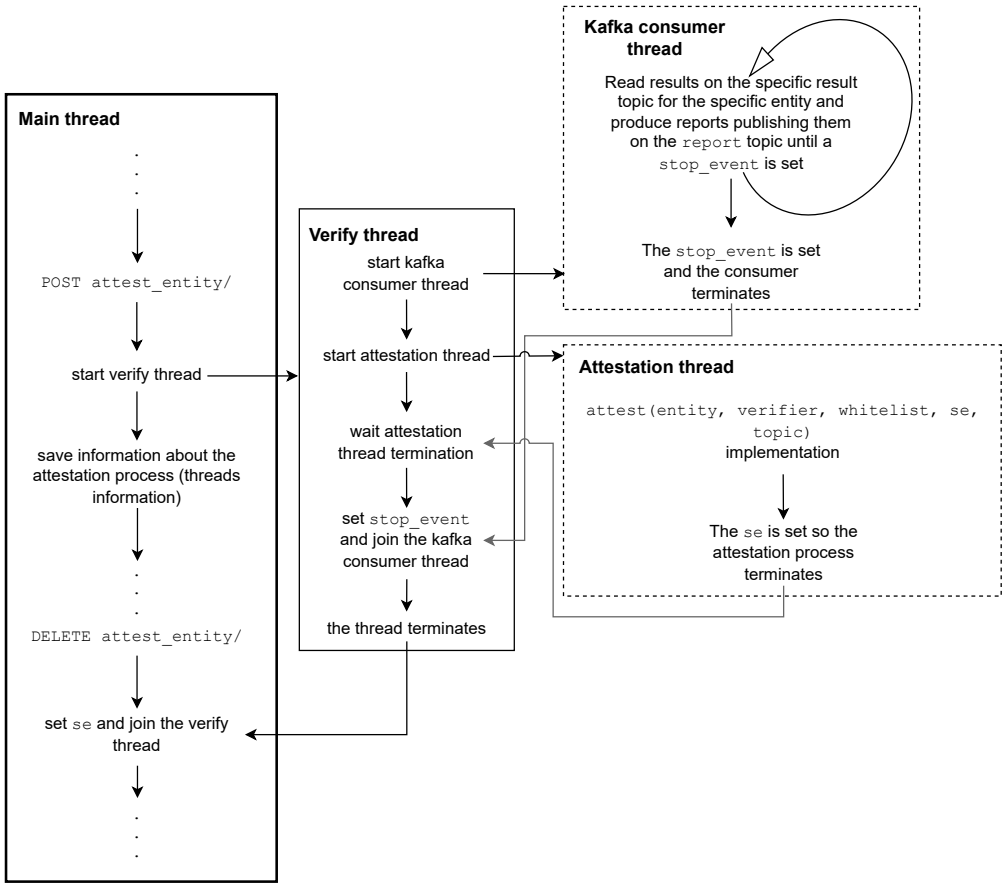

Figure 6.4. Attestation threads

- `/verifier`: It permits to manage the information stored in the Verifier database. It makes available three methods. When called with GET method it returns the information about the attestation technology specified in the body. In case it is called with method POST permits to add the information of a new attestation technology into the database. The last method allowed is the DELETE which permits to delete the information of the attestation technology specified.
- `/whitelist`: This API offers three methods. The GET method permits retrieving a single whitelist specified in the URL in the form `/whitelist?whitelist_uuid=<id>`. The POST method permits to store a new whitelist in the database. The DELETE method permits to delete from the database the specified whitelist.
- `/policy`: It also makes available the methods GET, POST and DELETE. The GET method permits retrieving from the database the policy related with a specific entity specified in the URL in the form `/policy?entity_uuid=<id>`. The POST method permits to store a new policy in the database. The DELETE method permits to delete a policy already present in the database.
- `/status`: This API permits knowing the current running remote attestation process in order to know which entities are under integrity verification.
- `/report`: This API permits retrieving reports included in a specific time interval.

Chapter 7

Test and Validation

The tests performed on the system can be divided into two categories: functional tests and performance tests. In order to perform the functional tests, it was used, as attestation technology, a patched version of the Keylime framework with support for remote attestation of pods, for which it has been developed a specific attestation adapter. In addition, to verify the correct behavior of the system, it was exploited a running instance of the containers orchestrator Kubernetes which has permitted to evaluate the possibility of attesting platforms (hosts) that contains subcomponents (pods).

7.1 Testbed

To evaluate the correct functioning of the proposed implementation, the testbed used was composed by:

- two attester machines which are Intel NUC equipped with an Intel Core i5-5300U Processor, 16 GB of RAM, and a TPM 2.0 chip, running the Keylime Agent component. The OS used is Ubuntu server 20.04 LTS with a patched Linux kernel based on version 5.13;
- one verifier machine which is an Intel NUC equipped with an Intel Core i5-5300U Processor, 16 GB of RAM, and a TPM 2.0 chip, running the Keylime Tenant, Registrar, and Verifier components. The OS used is Ubuntu server 20.04 LTS.
- one machine which is a DELL XPS 15 9500 equipped with an Intel Core i7-10750H Processor, 16 GB of RAM, running the Trust Monitor. The OS used is Ubuntu desktop 20.04 LTS.

7.2 Keylime attestation adapter

To perform these tests has been developed a specific attestation adapter for the Keylime framework version used. the APIs exposed by the component *Tenant Webapp* have been used to communicate with Keylime:

- POST `/agents/{agent_id:UUID}`

  This API permits to register a new Cloud Agent, identified by the `agent_id`, in the Cloud Verifier, in order to start the remote attestation. This API accepts several parameters to add a new agent. In the adapter developed the parameters used are:

  - `agent_ip` (string): indicates the IP address of the agent to add;
  - `ptype` (int) - *optional*: "payload type", it can have one of the following values: 0 = FILE (default), 1 = KEYFILE, 2 = CA DIR;

- `file_data` (string) - *optional*: it is the payload, base64 encoded, that is sent to the agent; if `ptype` = FILE, it will be encrypted by the Tenant with a random bootstrap key K_b; if `ptype` = KEYFILE, it is already encrypted with a K_b specified in the body attribute `keyfile_data`;
- `a_list_data` (list[string]) - *optional*: it is the whitelist of the host on which the agent runs;
- `e_list_data` (list[string]) - *optional*: it is the exclude list of the host which specify the files that have to be excluded from the system attestation. It is a list of regular expressions. Empty strings or strings starting with "`#`" will be removed from the list;
- `pods` (JSON object) - *optional*: it is the list of pods' information and for each pod contains as key the `uuid` of the pod and as value a JSON object containing a filed `a_list_data` which is a list of strings and represents the whitelist of the pod.

  Example body request:

```
{
    "agent_ip": "192.168.0.103",
    "ptype": 0,
    "file_data": "",
    "a_list_data": [
        "000...    boot_aggregate",
        "c168a1...3777f7   /bin/sh",
        "5945a4...3d72fb   /bin/gcc",
        ...
    ],
    "e_list_data":  [
        "/var/log/wtmp",
        "/root/etc/fstab",
        "/boot/grub/grubenv",
        "/sys/fs/.*",
        ...
    ],
    "pods": {
        "9be621ca-7746-4217-8a28-eab90077ac33": {
            "a_list_data": [
                "38ad97...52489b /pause",
                "1edf98...56d6b7 /usr/bin/local-path-provisioner"
            ]
        },
        "a94e4991-a53f-4952-87ee-6a2b0269e3bf": {
            "a_list_data": [
                "38ad97...52489b /pause",
                "d84ccc...2e4dae /metrics-server"
            ]
        }
    }
}

```

- GET `/agents/{agent_id:UUID}`

  This API permits getting the current state of a Cloud Agent, identified by the `agent_id`, to be able to determine its trustworthiness. It returns several parameters but in the proposed adapter the only field used is `operational_state`, which can have the following values:

  0 = REGISTERED, 1 = START, 2 = SAVED, 3 = GET QUOTE, 4 = GET QUOTE RETRY, 5 = PROVIDE V, 6 = PROVIDE V RETRY, 7 = FAILED, 8 = TERMINATED, 9 = INVALID QUOTE, 10 = TENANT FAILED.

The only value that permits to consider the host trusted is 3 (GET QUOTE). Of course, it is possible to check more parameters to make a more precise analysis.

- DELETE `/agents/{agent_id:UUID}`

  This API permits to remove a, identified by the `agent_id`, from the Cloud Verifier, in order to stop the remote attestation.

7.3 Functional tests

Functional tests are performed using the deployment of a Kubernetes cluster, composed of two worker nodes and one master node. The two worker nodes, in the Remote Attestation context, have the role of attester, and the master node has the role of the verifier. The process consists of the continuous remote attestation of the two attester nodes and the pods running in each node.

The first step was registering the Keylime framework in the Trust Monitor, in order to make available the information to contact its APIs. this registration has been performed calling the POST `/verfier` API with the following body request:

```
{
    "att_tech":"keylime_v6_3_2",
    "inf_id": 1,
    "metadata":{
        "tenant_ip":"192.168.0.114",
        "tenant_port":444
    }
}
```

Listing 7.1. Request body of Keylime verifier registration

Before registering nodes and pods, all the whitelists have been stored in the TM's database, in order to link them to entities at the registration moment. To store the whitelist it has been used the POST `/whitelist` API, with the following request body:

```
{
    "_id":2,
    "metadata":{
        "att_tech":"keylime_v6_3_2",
        "hash_algo":"sha265"
    },
    "whitelist":{
        "a_list_data":[
            "38ad97...2489b /usr/bin/ntfs-3g",
            "1edf98...6d6b7 /usr/bin/local-path-provisioner"
        ]
    }
}
```

Listing 7.2. Example of request body of whitelist storing

In this case, in listing 7.2, is reported only an example, because the actual request body, in the case of the node, has a significant dimension. This operation has been performed for each whitelist needed, both for nodes and pods. In this case, whitelists related to pods have been registered only for the first node, because the pods running on the two nodes are equal instances, so pods running on the second node had the same whitelists as the ones related to pods running on the first node.

Once performed the registration of the whitelists, all the nodes, and relative pods, have been registered, specifying the necessary information in the case of entity with type `node` or `pod`. In the case of nodes, the registration has been performed with the POST `/entity` API, sending the following body in the requests:

```
{
    "entity_uuid": 1,
    "inf_id": 1,
    "att_tech": ["keylime_v6_3_2"],
    "name": "attester",
    "external_id": "90e71d86-13e0-4bd3-9ec4-1521f10a5194",
    "type": "node",
    "whitelist_uuid": 1,
    "child": [2,3,4,5,6,7],
    "metadata": {
        "keylime_v6_3_2": {
            "agent_ip": "192.168.0.103",
            "e_list_data": ["^(?!/usr/bin/).*$"]
        }
    }
}
```

Listing 7.3. Request body of the first node registration

```
{
    "entity_uuid": 8,
    "inf_id": 1,
    "att_tech": ["keylime_v6_3_2"],
    "name": "attester_2",
    "external_id": "d432fbb3-d2f1-4a97-9ef7-75bd81c00000",
    "type": "node",
    "whitelist_uuid": 7,
    "child": [9,10,11,12,13,14,15,16,17,18,19],
    "metadata": {
        "keylime_v6_3_2": {
            "agent_ip": "192.168.0.100",
            "e_list_data": ["^(?!/usr/bin/).*$"]
        }
    }
}
```

Listing 7.4. Request body of the second node registration

the `external_id` represents for both nodes the identifier in the Keylime framework. In the `metadata` field has been declared some information that is used inside the Keylime adapter. The attribute `agant_ip` is sent by the adapter to the Keylime verifier. In this way, the verifier can contact the agent in order to start the remote attestation process. The attribute `e_list_data` represents the exclude list that can be specified in order to exclude some paths from the integrity verification.

Calling the same API all the pods running on each node have been registered. The registration of a pod required less information, which is reported below:

```
{
    "entity_uuid": 19,
    "att_tech": ["keylime_v6_3_2"],
    "name": "svclb",
    "external_id": "dd5e909a-f74a-407d-99a5-1f97020099b8",
    "type": "pod",
    "whitelist_uuid": 6,
    "parent": 8,
    "metadata": {}
}
```

Listing 7.5. Request body of one of `svclb` pods registration

In this case, some information is not necessary, like the `inf_id` field, because this kind of object (`pod`) cannot be attested directly, in the case of Keylime, but there is the necessity to attest the entire node. For this reason, the management of these objects is performed by the Keylime adapter, because it is strictly related to the framework workflow.

After having registered all this data, it has been started the remote attestation process, calling the POST `/attest_entity` API for each node with their respective bodies:

```
{
    "entity_uuid": 1
}
```

```
{
    "entity_uuid": 8
}
```

receiving as a response, for each request, the remote attestation successfully started:

```
{
    "Message": "entity 1 - attestation thread started successfully"
}
```

```
{
    "Message": "entity 8 - attestation thread started successfully"
}
```

It has been checked the state of the TM, after having started the remote attestation, and the result was the following:

```
{
    "adapters_loaded":[
        "keylime_v6_3_2"
    ],
    "att_processes":[
        {
            "att_tech":[
                "keylime_v6_3_2"
            ],
            "entity_uuid":1,
            "external_id":"90e71d86-13e0-4bd3-9ec4-1521f10a5194",
            "name":"attester"
        },
        {
            "att_tech":[
                "keylime_v6_3_2"
            ],
            "entity_uuid":8,
            "external_id":"d432fbb3-d2f1-4a97-9ef7-75bd81c00000",
            "name":"attester_2"
        }
    ]
}
```

where:

- `"adapters_loaded"` contains the list of adapters currently loaded by the TM;
- `"att_processes"` contains the list of active attestation processes;
- `"att_tech"` contains the list of attestation technologies that are attesting the entity;

- `"entity_uuid"` represents the entity identifier;
- `"external_id"` is the identifier used inside the Keylime attestation adapter;
- `"name"` is the name of the entity.

Once started the remote attestation processes on the entities, the TM core modifies their state in the database, that change from `registered` to `attesting`, and it has been verified by calling the GET `/entity?entity_uuid=1` and `/entity?entity_uuid=8` receiving as responses:

```
{
    "entity_uuid": 1,
    "inf_id": 1,
    "att_tech": ["keylime_v6_3_2"],
    "name": "attester",
    "external_id": "90e71d86-13e0-4bd3-9ec4-1521f10a5194",
    "type": "node",
    "whitelist_uuid": 1,
    "child": [2,3,4,5,6,7],
    "parent": null,
    "state": "attesting",
    "metadata": {
        "keylime_v6_3_2": {
            "agent_ip": "192.168.0.103",
            "e_list_data": ["^(?!/usr/bin/).*$"]
        }
    }
}
```

```
{
    "entity_uuid": 8,
    "inf_id": 1,
    "att_tech": ["keylime_v6_3_2"],
    "name": "attester_2",
    "external_id": "d432fbb3-d2f1-4a97-9ef7-75bd81c00000",
    "type": "node",
    "whitelist_uuid": 7,
    "child": [9,10,11,12,13,14,15,16,17,18,19],
    "parent": null,
    "state": "attesting",
    "metadata": {
        "keylime_v6_3_2": {
            "agent_ip": "192.168.0.100",
            "e_list_data": ["^(?!/usr/bin/).*$"]
        }
    }
}
```

To read the aggregate reports has been developed a dummy consumer that permits to verify the reports are correctly published on the queue. This consumer read the queue with a 5 seconds timeout in case of empty queue and an example of the output can be the following:

```
{
    "entity_uuid": 8,
    "trust": true,
    "state": [
        {
            "att_tech": "keylime_v6_3_2",
            "trust": true
```

```
        }
    ],
    "time": "2022-09-08 11:08:17.979403"
}

{
    "entity_uuid": 8,
    "trust": true,
     "state": [
        {
            "att_tech": "keylime_v6_3_2",
            "trust": true
        }
    ],
    "time": "2022-09-08 11:08:38.187520"
}

...
```

where:

- `"entity_uuid"` represents the entity identifier;
- `"trust"` indicates the trustworthiness of the entity, based on attestation results received from all attestation technology that are attesting the entity;
- `"state"` contains the list of attestation results, one for each attestation technology that is attesting the entity;
- `"att_tech"` indicates a specific attestation technology;
- `"trust"` represents the attestation result for a specific technology;
- `"time"` indicates the time at which has been created the report.

A host becomes untrusted, for example, if an unexpected script, so not present in the whitelist, is executed:

```
$ echo "#!/bin/bash\necho "Hacked!"" > malicious_script.sh
$ chmod +x malicious_script.sh
$ ./malicious_script.sh
```

In this case, the host becomes untrusted, so the `operationl_state` returned to the attestation adapter is 9 = INVALID QUOTE and the report becomes the following one:

```
{
    "entity_uuid": 8,
    "trust": false,
    "state": [
        {
            "att_tech": "keylime_v6_3_2",
            "trust": false
        }
    ],
    "time": "2022-09-08 12:34:15.852947"
}
...
```

In order to stop the remote attestation processes it has been called, for each host, the DELETE `/attest_entity` API specifying the entity in the body:

```
{
    "entity_uuid": 1
}
```

```
{
    "entity_uuid": 8
}
```

receiving as a response, for each request, the remote attestation successfully stopped:

```
{
    "Message": "entity 1 - attestation stopped successfully"
}
```

```
{
    "Message": "entity 8 - attestation stopped successfully"
}
```

7.4 Performance tests

The tests performed to evaluate the solution's performance are focused on the following metrics: time taken by the Trust Monitor to start a new attestation process, depending on the whitelist dimension; time taken for the registration of a new entity, depending on the whitelist dimension; CPU utilization and RAM utilization depending on the number of attestation processes running.

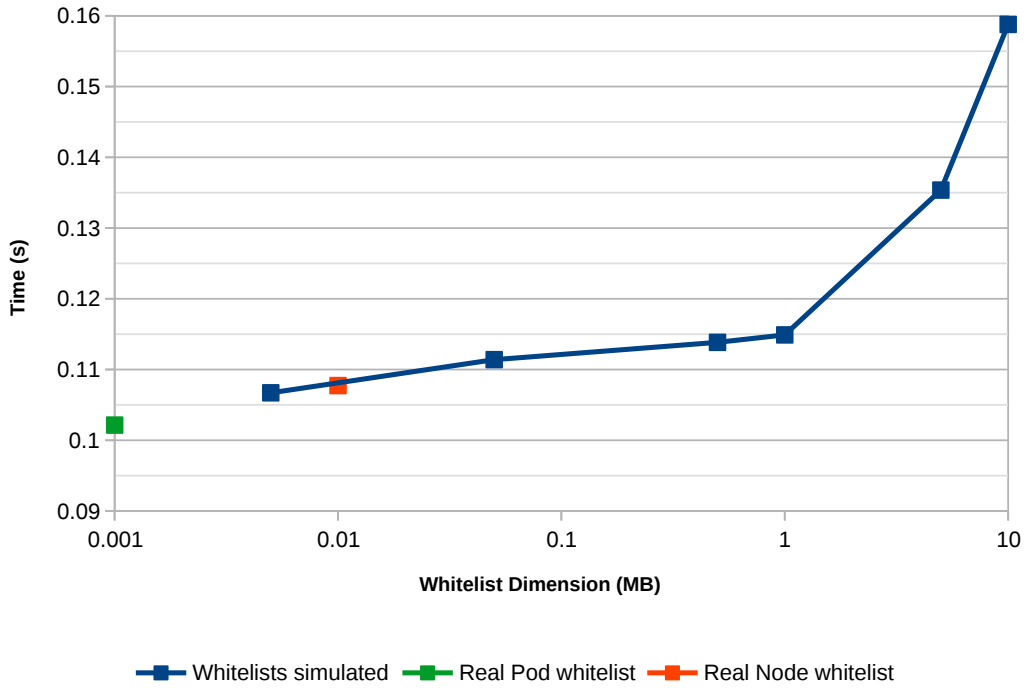

Figure 7.1. Time taken by the TM to start attestation process depending on whitelist dimension

Figure 7.1 shows how the time required to start the attestation process change using whitelists of an increasing dimension. This measure represents how much time is used by the TM core to retrieve the information needed and starts all threads in order to begin the attestation process. The tested dimensions are 5kB, 50kB, 500kB, 1MB, 5MB and 10 MB. In the figure are also

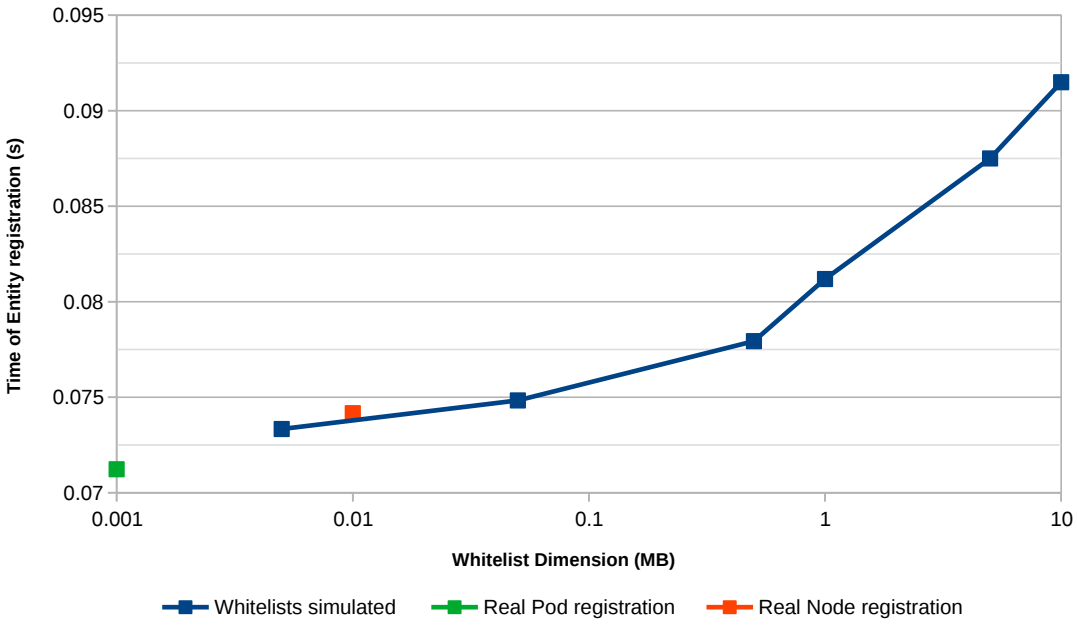

Figure 7.2. Time taken by the TM to start attestation process depending on whitelist dimension

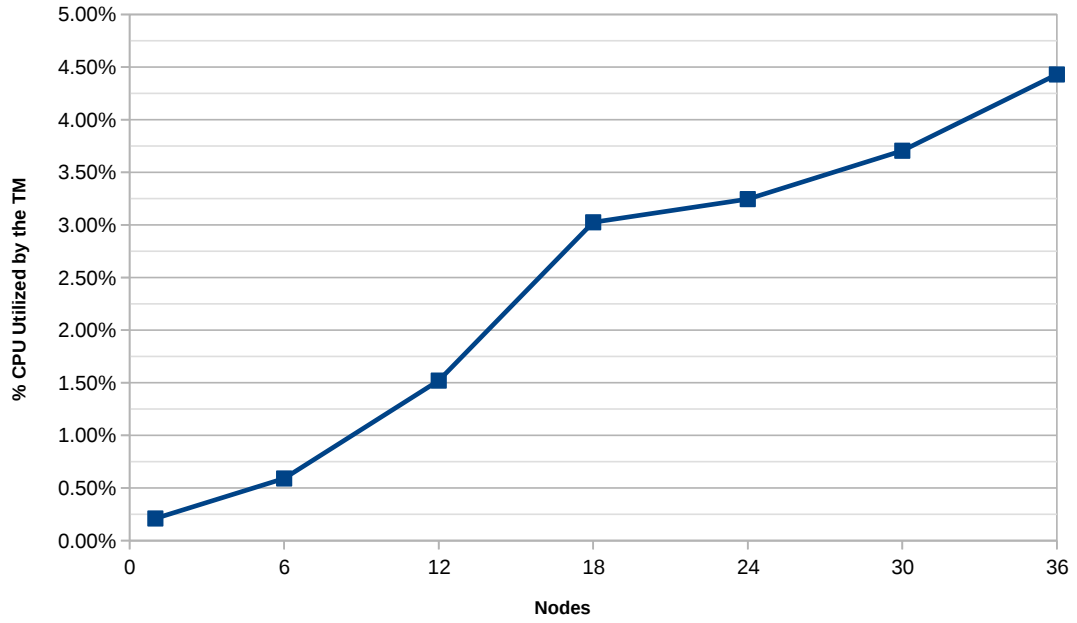

Figure 7.3. TM CPU consumption depending on the number of attested nodes

reported the results obtained using two real cases: a node whitelist and a pod whitelist. The results show that the time starts to significantly increase when the whitelist dimension becomes bigger than 1 MB.

Figure 7.2 shows the registration time depending on the dimension of the whitelist associated. The tested dimensions are the same as in the previous case. The results show an almost linear behavior of the time of entity registration.

In order to evaluate CPU and RAM consumption because of attestation processes managed

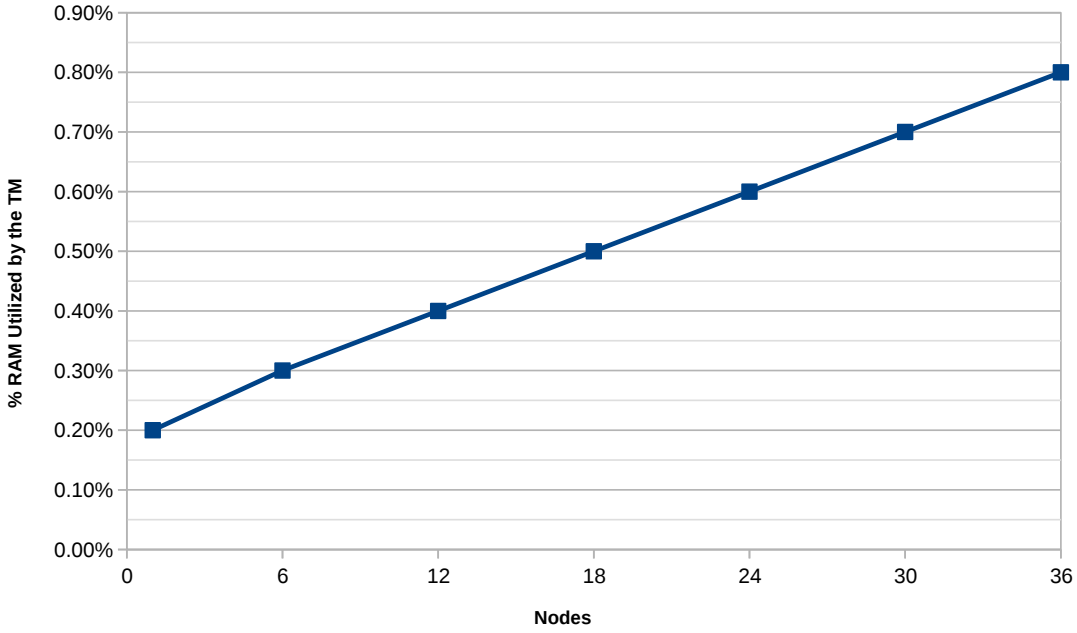

Figure 7.4. TM RAM consumption depending on the number of attested nodes

by the TM, it has been deployed a test environment. This environment is composed of six attestation technologies (infrastructures) simulated running in Docker containers. These attestation technologies simulate the behavior of the Keylime framework, in order to be able to use the attestation adapter developed. Six nodes have been registered for each infrastructure.

Figure 7.3 shows CPU consumption depending on the number of nodes under attestation. The results show an almost linear increase in the CPU percentage used.

The figure 7.4 shows the memory consumption which has a linear behavior in relation to the number of nodes under attestation.

From the data obtained, it is possible to notice that the proposed solution is scalable in relation to the number of entities under attestation.

Chapter 8

Conclusions and future work

The main objective of this thesis was to introduce a solution that can define a general model for remote attestation in order to be able to attest different objects. Another objective was that this solution has to be able to manage different attestation technologies and aggregate their results. In addition, it has to be able to manage several attestation processes of multiple entities in order to be integrable in cloud environments. In order to reach these goals, the solution proposed consists of a redesign and reimplementation of the Trust Monitor.

The new design was necessary to make the TM completely independent from the attestation technologies used to perform remote attestation on the objects that compose the infrastructure. This was possible by introducing an Adapter's Connector, which permits to dynamically load several drivers, called Adapters, which implement the interface with different attestation technologies. In this way, it is possible to add an attestation technology developing the corresponding adapter and add it to the TM which will use it in case an entity requires that particular technology. An adapter must respect a fixed structure in order that the correctors can understand it. This solution moves the most of attestation logic on the adapter because it implements the specific process to interact with the corresponding attestation framework. The fixed structure of attestation adapters also permits to identify a specific attestation process that is able to abstract the underlying technology, so providing a general model applicable independently from the framework used to perform the specific remote attestation process.

In order to be able to register several kinds of objects into the TM database, it was designed a specific schema that permits to specify all the information needed to start the remote attestation. In addition, some variable fields have been added which permit the user to add some custom information (metadata) used by the developed adapters. In this way, even if the structure of an entity is fixed, a lot of flexibility is added. This flexibility is available also in the case of whitelists, because it is possible to register a whitelist in any form needed, and it will be processed by the attestation adapter of the right attestation framework. Also in the case of the whitelists, there is the possibility to store some metadata which permits to add custom information.

To test the system it has been developed an attestation adapter a patched version of the remote attestation framework Keylime. This particular version has the support for remote attestation of nodes and Kubernetes' pods, which has permitted to perform the remote attestation of nested objects, allowing to verify the objects model designed. In addition, it has been developed a simulated environment in order to test the management of the attestation of different entities belonging to different infrastructures. To realize this environment it has been created a simulation of the Keylime framework, which has been deployed as a container permitting to run more than one instance in order to be able to simulate the presence of multiple infrastructures. This has also permitted to perform some performance tests on the solution, that show how the system behaves in different situations. In particular, tests show that the system is scalable in terms of attestation processes managed at the same time, and the overhead added by the TM to start the attestation process remains acceptable even if the whitelists used have dimensions of the order of 10 MB.

The proposed solution can be improved by adding a system of automatic generation of

whitelists that can infer them for nodes, containers, pods, etc. In addition, it can update them when the software is updated. Another improvement could be brought by introducing an external interface for the TM which can be command-line based, graphical or an integration for some infrastructure orchestrator like Kubernates. They can also be performed other tests that can help to verify and improve the possibility to use TEE technologies, such as Intel SGX, AMD SEV and ARM TrustZone. This solution has many possibilities for expansion and improvement that can allow it to be integrated into emerging IT infrastructure scenarios.

Bibliography

[1] R. Yeluri and E. Castro-Leon, "Building the infrastructure for cloud security: A solutions view", Springer Nature, 2014

[2] T. T. Brooks, C. Caicedo, and J. S. Park, "Security vulnerability analysis in virtualized computing environments", International Journal of Intelligent Computing Research, vol. 3, December 2012, pp. 277–291, DOI 10.20533/ijicr.2042.4655.2012.0034

[3] V. Varadharajan and U. Tupakula, "Security as a service model for cloud environment", IEEE Transactions on Network and Service Management, vol. 11, March 2014, pp. 60–75, DOI 10.1109/TNSM.2014.041614.120394

[4] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation", International Journal of Information Security, vol. 10, June 2011, pp. 63–81, DOI 10.1007/s10207-011-0124-7

[5] TCG, "TPM main part 1 design principles", TCG White Paper, 2011

[6] G. ETSI, "Network Functions Virtualisation (NFV); Security; Report on NFV Remote Attestation Architecture", ETSI GR NFV-SEC 018 V1.1.1, November 2019. https://www.etsi.org/deliver/etsi_gr/NFV-SEC/001_099/018/01.01.01_60/gr_NFV-SEC018v010101p.pdf

[7] The Kubernetes project, https://kubernetes.io/

[8] T. Su, A. Lioy, and N. Barresi, "Trusted computing technology and proposals for resolving cloud computing security problems", Cloud Computing Security, pp. 371–384, CRC Press, 2016, DOI 10.1201/9781315372112-41

[9] S. W. Smith, "Trusted computing platforms: design and applications", Springer, 2013

[10] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. Van Doorn, "A practical guide to trusted computing", Pearson Education, 2007

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

[12] TCG, "Trusted platform module library part 1: Architecture", 2018

[13] I. Sfyrakis and T. Gross, "A survey on hardware approaches for remote attestation in network infrastructures", arXiv preprint arXiv:2005.12453, May 2020, DOI 10.48550/arXiv.2005.12453

[14] R. V. Steiner and E. Lupu, "Attestation in wireless sensor networks: A survey", ACM Comput. Surv., vol. 49, September 2016, DOI 10.1145/2988546

[15] L. Chen and B. Warinschi, "Security of the tcg privacy-ca solution", 2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, 2010, pp. 609–616, DOI 10.1109/EUC.2010.98

[16] E. Brickell, J. Camenisch, and L. Chen, "Direct Anonymous Attestation", Proceedings of the 11th ACM Conference on Computer and Communications Security, New York (NY, USA), 2004, pp. 132–145, DOI 10.1145/1030083.1030103

[17] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn, "Design and implementation of a TCG-based integrity measurement architecture", USENIX Security symposium, San Diego (CA, USA), August 2004, pp. 223–238

[18] D. Safford and M. Zohar, "Trusted computing and open source", Information Security Technical Report, vol. 10, August 2005, pp. 74–82, DOI 10.1016/j.istr.2005.05.001

[19] S. Schulz, A.-R. Sadeghi, and C. Wachsmann, "Short paper: Lightweight remote attestation using physical functions", Proceedings of the Fourth ACM Conference on Wireless Network Security, Hamburg (Germany), June 2011, pp. 109–114, DOI 10.1145/1998412.1998432

[20] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems", Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, Brighton (United Kingdom), December 2005, pp. 1–16, DOI 10.1145/1095810.1095812

[21] K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "Hydra: Hybrid design for remote attestation (using a formally verified microkernel)", Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks, Boston (MA, USA), July 2017, pp. 99–110, DOI 10.1145/3098243.3098261

[22] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "Smart: secure and minimal architecture for (establishing dynamic) root of trust", Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks, Boston (MA, USA), July 2017, pp. 99–110

[23] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments", Proceedings of the Fifteenth European Conference on Computer Systems, Heraklion (Greece), April 2020, pp. 1–6, DOI 10.1145/3342195.3387532

[24] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution", Hasp@ isca, vol. 10, June 2013, DOI 10.1145/2487726.2488368

[25] V. Costan and S. Devadas, "Intel SGX Explained." Cryptology ePrint Archive, Paper 2016/086, 2016, https://eprint.iacr.org/2016/086

[26] TCG, "Hardware requirements for a device identifier composition engine", TCG Published, 2018

[27] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base", 22nd USENIX Security Symposium (USENIX Security 13), Washington D.C. (USA), August 2013, pp. 479–498

[28] L. Jäger, R. Petri, and A. Fuchs, "Rolling dice: Lightweight remote attestation for cots iot hardware", Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria (Italy), August 2017, DOI 10.1145/3098954.3103165

[29] P. England, A. Marochko, D. Mattoon, R. Spiger, S. Thom, and D. Wooten, "Riot - a foundation for trust in the internet of things", Tech. Rep. MSR-TR-2016-18, April 2016. https://www.microsoft.com/en-us/research/publication/riot-a-foundation-for-trust-in-the-internet-of-things/

[30] H. Birkholz, M. Wiseman, H. Tschofenig, N. Smith, and M. Richardson, "Remote Attestation Procedures Architecture", Internet-Draft draft-birkholz-rats-architecture-03, Internet Engineering Task Force, November 2019. Work in Progress https://datatracker.ietf.org/doc/draft-birkholz-rats-architecture/03/

[31] H. Birkholz, D. Thaler, M. Richardson, N. Smith, and W. Pan, "Remote Attestation Procedures Architecture", Internet-Draft draft-ietf-rats-architecture-18, Internet Engineering Task Force, June 2022. Work in Progress https://datatracker.ietf.org/doc/draft-ietf-rats-architecture/18/

[32] M. Pei, H. Tschofenig, D. Wheeler, A. Atyeo, and D. Liu, "Trusted Execution Environment Provisioning (TEEP) Architecture", Internet-Draft draft-ietf-teep-architecture-01, Internet Engineering Task Force. Work in Progress https://datatracker.ietf.org/doc/draft-ietf-teep-architecture/01/

[33] M. Pei, H. Tschofenig, D. Thaler, and D. Wheeler, "Trusted Execution Environment Provisioning (TEEP) Architecture", Internet-Draft draft-ietf-teep-architecture-17, Internet Engineering Task Force, April 2022. Work in Progress https://datatracker.ietf.org/doc/draft-ietf-teep-architecture/17/

[34] G. ETSI, "Network Functions Virtualisation (NFV): Architectural Framework", ETSI GS NFV 002 v1.1.1, October 2013. https://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.01.01_60/gs_NFV002v010101p.pdf

[35] G. ETSI, "Network Functions Virtualisation (NFV); NFV Security; Security and Trust Guidance", ETSI GR NFV-SEC 003 V1.2.1, August 2016. https://www.etsi.org/deliver/etsi_gr/NFV-SEC/001_099/003/01.02.01_60/gr_NFV-SEC003v010201p.pdf

[36] M. De Benedictis and A. Lioy, "A proposal for trust monitoring in a network functions

virtualisation infrastructure", 2019 IEEE Conference on Network Softwarization (NetSoft), June 24-28, 2019, pp. 1–9, DOI 10.1109/NETSOFT.2019.8806655
[37] The Docker project, https://www.docker.com/
[38] M. De Benedictis and A. Lioy, "Integrity verification of docker containers for a lightweight cloud environment", Future Generation Computer Systems, vol. 97, August 2019, pp. 236–246, DOI 10.1016/j.future.2019.02.026
[39] N. Schear, P. T. Cable, T. M. Moyer, B. Richard, and R. Rudd, "Bootstrapping and maintaining trust in the cloud", Proceedings of the 32nd Annual Conference on Computer Security Applications, December 2016, pp. 65–77, DOI 10.1145/2991079.2991104
[40] TCG, "Attestation key identity certification", Trusted Platform Module Library Part 1: Architecture, pp. 28–29, TCG Published, November 2019
[41] The Python project, https://www.python.org/
[42] Introduction to asyncio, https://pgjones.gitlab.io/quart/tutorials/asyncio.html#asyncio
[43] The Apache Kafka Project, https://kafka.apache.org/
[44] The JSON Schema specification for Python, https://json-schema.org/

Appendix A

User's Manual

A.1 System deployment

The entire system can be deployed as a set of Docker containers using the tool `docker-compose`. The first requirement is to install Docker Engine following the instructions specified in the reference documentation of the Docker official website https://docs.docker.com/engine/install/ubuntu/. In addition, it is required also to install `docker-compose` following the documentation on the official website https://docs.docker.com/compose/install/.

In order to be able to reach a correct deployment, it is needed to build the Docker image of the Trust Monitor. To do so, it is sufficient to use the `Dockerfile` present in the main project directory. The procedure to build the image is the following:

1. Move in the main directory of the project `trust-monitor`;
2. execute the following command:

   ```
   $ docker build -t trust-monitor:<tag> .
   ```

   where `<tag>` is the version of the current image that is being built;
3. move into the `docker-compose` subdirectory;
4. modify the `trust-monitor` section in the `docker-compose.yml` file substituting the previous version (under the `image` label) with the new one;
5. remaining in the `docker-compose` subdirectory launch the command:

   ```
   $ docker-compose -f docker-compose.yml up -d
   ```
6. if it is the **first execution** of the system, execute the `init-docker.sh` script in order to initialize databases:

   ```
   $ ./init-docker.sh
   ```

Once completed this procedure, the Trust Monitor will be available on the `localhost` on port `5000`.

The system can be deployed running the Trust Monitor, not in a docker container. In this case the section `trust-monitor`, in the `docker-compose.yml` file, has to be commented. At his point, inside the `docker-compose` directory, launch the command:

```
$ docker-compose -f docker-compose.yml up -d
```

and in case of the first execution run also the command:

```
$ ./init-docker.sh
```

Once started the containers, it is necessary to set the environment variable:

```
QUART_APP=api-manager:app
```

and then move into the main directory and run the command:

```
$ quart run
```

Once completed this procedure, the Trust Monitor will be available on the `localhost` on port `5000`.

A.1.1 Enabling TLS

In order to enable TLS on the API Manager it is sufficient to uncomment the `tls` section in the configuration file `config.ini` and modify the three associated parameters:

- `ca_certs="path/of/CA/certificate"`
- `certfile="path/of/certificate"`
- `keyfile="path/of/key"`

These files need to be added to the project directory in order to be copied during the docker image build.

A.1.2 Keylime

In case there is a necessity to install the Keylime framework, it can be done following the documentation on the official website https://keylime.dev/.

A.1.3 Adding attestation adapters

In order to be able to add a new attestation adapter, it has to be developed following some constraints.

The structure of the adapter must be a single python file and the name has to be the same as the attestation technology name that will be used in the TM (e.g. if the attestation technology name is `keylimev6_3_2` the file name must be `keylimev6_3_2.py`).

The file must contain one class with a name that can be chosen by the developer. The class must contain four methods:

1. `register(entity, whitelist, verifier)`

   This method is called when a new entity is registered in the TM, after having saved it in the TM database. It receives three parameters. The first one is `entity` (listing A.1) which contains all the information about the new entity to register.

```
{
    "entity_uuid": 1,
    "inf_id": 1,
    "att_tech": ["keylime_v6_3_2"],
    "name": "attester",
    "external_id": "90e71d86-13e0-4bd3-9ec4-1521f10a5194",
    "type": "node",
    "whitelist_uuid": 1,
    "child": [2,3,4,5,6,7],
    "parent": null,
    "state": "registered",
    "metadata": {
        "keylime_v6_3_2": {
```

```
            "agent_ip": "192.168.0.103",
            "e_list_data": ["^(?!/usr/bin/).*$"]
        }
    }
}

```

Listing A.1. `entity` object example

The second one is `whitelist` (listing A.2) which contains all the information about the whitelist linked with the object.

```
{
    "_id":2,
    "metadata":{
        "att_tech":"keylime_v6_3_2",
        "hash_algo":"sha265"
    },
    "whitelist":{
        "a_list_data":[
            "38ad97...2489b /pause",
            "1edf98...6d6b7 /usr/bin/local-path-provisioner"
        ]
    }
}

```

Listing A.2. `whitelist` object example

The third object is `verifer` (listing A.3) and contains all the information about the verifier to contact.

```
{
    "att_tech":"keylime_v6_3_2",
    "inf_id": 1,
    "metadata":{
        "tenant_ip":"192.168.0.114",
        "tenant_port":444
    }
}

```

Listing A.3. `verifier` object example

All the `metadata` fields are defined by the user so the semantics of those fields is defined by the administrator.

2. `attest(entity, verifier, whitelist, se, topic)`

   This method is called when the POST `/attest_entity` API is called. This method is executed in a dedicated thread. The `entity`, `whitelist` and `verifier` parameters are the same of the previous case. The `se` parameter is a Threading Event which permits to maintain the thread running until the DELETE `/attest_entity` API is called. After this call the `se` will be set and the condition `se.is_set()` will become true. The `topic` parameter is a string and represents the topic on which publish the attestation results received from the attestation framework.

3. `delete(entity, verifier)`

   This method is called when an entity is deleted from the TM database and permits to delete the object from the attestation framework. The parameters are the same as in the previous cases.

4. `status(verifier)`

   This method permits obtaining information about the current state of the remote attestation framework. The `verifier` parameter is the same as in the previous cases.

Once developed the new attestation adapter, in order to make it available in the TM it needs to be placed in the `adapters` subdirectory of the project. After having added the file to this subdirectory, it has to be declared in the configuration file `config.ini` in the `adapters` section. The line to add is:

```
<name of the attestation technology> = <name of the class implemented>
```

in this way it will be possible to dynamically load the new adapter in the TM logic.

After having added the new adapter it will be needed to execute the procedure to build a new image of the Trust Monitor in order to deploy the system as explained in paragraph A.1.

A.2 Use of databases

All the databases have some attributes that have fixed semantics because they are used inside the TM logic. Some other attributes have flexible semantics which means that the administrator or the attestation adapters' developer can in order to can use those values in new adapters with the needed semantics.

Instances database

In the Instances database, there are eight attributes with a fixed meaning that can be changed by the administrator. These attributes are:

- `entity_uuid`;
- `type`;
- `inf_id`;
- `name`;
- `att_tech`;
- `whitelist_uuid`;
- `child`;
- `parent`.

These attributes are used inside the TM logic so they have a specified role that cannot be changed as explained in paragraph 5.3.3. The other attributes can be used by the administrator or the attestation adapters' developer to store some useful information needed in the remote attestation process. Attributes with free semantics are `external_id`, and `metadata` where the content and the meaning of these fields can be decided by the administrator. The field `metadata`, in the current implementation, has the only constraint to be divided into different sub-objects, where each one represents the metadata for a single attestation technology. The content of each section is free to be decided by the administrator or the developer. The `type` field has fixed semantics but it is not used directly in the TM core logic, so it can be used in several ways depending on the needs.

Whitelist database

In this case, the only attribute with specific semantics is `_id` which must contain an integer to identify the whitelist. The `metadata` field can contain any information that could be useful during the whitelist utilization. The `whitelist` field is the most important one which contains the actual data about the whitelist. This field can have a custom structure depending on the needs of the attestation technology.

Verfiers database

In this case, the attributes are three and only the `metadata` field can have custom content and structure. The `att_tech` must contain the name of the attestation technology according to the name of the corresponding adapter. The `inf_id` must contain the identifier of the infrastructure to which the attestation framework refers. The `metadata` must be a JSON object which can have a custom structure and contains custom information needed.

Appendix B

Developer's reference guide

B.1 Trust Monitor APIs

This section describes the Trust Monitor APIs made available in order to control the system and the remote attestation processes.

GET `/entity?entity_uuid=<id>`

Get the information about an entity registered in the TM or about all entities registered.

Response JSON object

- `entity_uuid` (int): Identifier of the object in the TM's database;
- `inf_id` (int): The identifier of the infrastructure;
- `att_tech` (string[]): List of all attestation technologies which perform the remote attestation on the object;
- `name` (string): Name of the entity;
- `external_id` (string): External identifier set during registration;
- `type` (string): Type of the entity;
- `whitelist_uuid` (int): The identifier of the entity whitelist;
- `child` (int[]): List of the entities contained in the specified one (it could be empty).
- `parent` (int): The identifier of the entity that contains the specified one (it could be `null`);
- `state` (string): The state of the entity in the TM system;
- `metadata` (json): The JSON object containing all metadata about the entity.

Example response:

```
{
    "entity_uuid": 1,
    "inf_id": 1,
    "att_tech": ["keylime_v6_3_2"],
    "name": "attester",
    "external_id": "90e71d86-13e0-4bd3-9ec4-1521f10a5194",
```

```
    "type": "node",
    "whitelist_uuid": 1,
    "child": [2,3,4,5,6,7],
    "parent": null,
    "state": "registered",
    "metadata": {
        "keylime_v6_3_2": {
            "agent_ip": "192.168.0.103",
            "e_list_data": ["^(?!/usr/bin/).*$"]
        }
    }
}
```

In case of failure the returned code depends on the cause of the failure and the response body will communicate the error occurred:

```
{
    "Error": "This string will report the error occurred"
}
```

In case no `entity_uuid` is provided in the URL, the response will contain the whole list of entitties registered in the TM.

Example response:

```
{
    "entities": [
        {
            "entity_uuid": 1,
            "inf_id": 1,
            "att_tech": ["keylime_v6_3_2"],
            ...
        },
        {
            "entity_uuid": 2,
            "inf_id": 1,
            "att_tech": ["keylime_v6_3_2"],
            ...
        },
        ...
    ]
}
```

POST `/entity`

Register a new entity in the TM.

Request JSON object

- `entity_uuid` (int): Identifier of the object in the TM's database;
- `inf_id` (int): The identifier of the infrastructure;
- `att_tech` (string[]): List of all attestation technologies which perform the remote attestation on the object;
- `name` (string): Name of the entity;
- `external_id` (string): Custom identifier;

- `type` (string): Type of the entity;
- `whitelist_uuid` (int): The identifier of the entity whitelist;
- `child` (int[]): List of the entities contained in the specified one (it could be empty).
- `parent` (int): The identifier of the entity that contains the specified one (it could be `null`);
- `metadata` (json): The JSON object containing all metadata about the entity (it is custom information that can be needed during the process of remote attestation).

Example request:

```
{
    "entity_uuid": 1,
    "inf_id": 1,
    "att_tech": ["keylime_v6_3_2"],
    "name": "attester",
    "external_id": "90e71d86-13e0-4bd3-9ec4-1521f10a5194",
    "type": "node",
    "whitelist_uuid": 1,
    "child": [2,3,4,5,6,7],
    "metadata": {
        "keylime_v6_3_2": {
            "agent_ip": "192.168.0.103",
            "e_list_data": ["^(?!/usr/bin/).*$"]
        }
    }
}
```

Response JSON object

If the registration is completed successfully the TM returns `200` with a response body like the following:

```
{
    "Message": "entity 1 successfully registered"
}
```

In case of failure the returned code depends on the cause of the failure and the response body will communicate the error occurred:

```
{
    "Error": "This string will report the error occurred"
}
```

PUT /entity

Edit an entity registered in the TM.

Request JSON object

In this case, the parameters to send in the request body are all optional. In this way, values that will be modified are only those in the request body.

- `entity_uuid` (int): Identifier of the object in the TM's database. This is the only value that **cannot be modified** but it is necessary in order to identify the specific entity;
- `att_tech` (string[]): List of all attestation technologies which perform the remote attestation on the object;

- `name` (string): Name of the entity;
- `external_id` (string): Custom identifier;
- `type` (string): Type of the entity;
- `whitelist_uuid` (int): The identifier of the entity whitelist;
- `child` (int[]): List of the entities contained in the specified one (it could be empty).
- `parent` (int): The identifier of the entity that contains the specified one (it could be `null`);
- `metadata` (json): The JSON object containing all metadata about the entity (it is custom information that can be needed during the process of remote attestation).

Example request:

```
{
    "entity_uuid": 1,
    "whitelist_uuid": 3,
    "child": [2,3,4],
    "metadata": {
        "keylime_v6_3_2": {
            "agent_ip": "192.168.0.173",
            "e_list_data": ["^(?!/usr/bin/).*$"]
        }
    }
}
```

Response JSON object

If the modification is completed successfully the TM returns `200` with a response body like the following:

```
{
    "Message": "entity 1 successfully updated"
}
```

In case of failure the returned code depends on the cause of the failure and the response body will communicate the error occurred:

```
{
    "Error": "This string will report the error occurred"
}
```

DELETE /entity

Delete an entity registered in the TM.

Request JSON object

- `entity_uuid` (int): Identifier of the object in the TM's database.

Example request:

```
{
    "entity_uuid": 1
}
```

Response JSON object

If the delete is completed successfully the TM returns `200` with a response body like the following:

```
{
    "Message": "entity 1 successfully deleted"
}
```

In case of failure the returned code depends on the cause of the failure and the response body will communicate the error occurred:

```
{
    "Error": "This string will report the error occurred"
}
```

GET /verifier

Get the information about an attestation technology registered in the TM.

Request JSON object

- `att_tech` (string): The name of the attestation technology;
- `inf_id` (int): The identifier of the infrastructure.

Example request:

```
{
    "att_tech": "keylime_v6_3_2",
    "inf_id": 1
}
```

Response JSON object

- `att_tech` (string): The name of the attestation technology;
- `inf_id` (int): The identifier of the infrastructure;
- `metadata` (json): The JSON object containing all metadata about the attestation technology.

Example response:

```
{
    "att_tech":"keylime_v6_3_2",
    "inf_id": 1,
    "metadata":{
        "tenant_ip":"192.168.0.114",
        "tenant_port":444
    }
}
```

POST /verifier

Register a new attestation technology in the TM.

Request JSON object

- `att_tech` (string): The name of the attestation technology;
- `inf_id` (int): The identifier of the infrastructure;
- `metadata` (json): The JSON object containing all metadata about the attestation technology.

Example request:

```
{
    "att_tech":"keylime_v6_3_2",
    "inf_id": 1,
    "metadata":{
        "tenant_ip":"192.168.0.114",
        "tenant_port":444
    }
}
```

Response JSON object

If the registration is completed successfully the TM returns `200` with a response body like the following:

```
{
    "Message": "verifier keylime_v6_3_2 successfully registered"
}
```

In case of failure the returned code depends on the cause of the failure and the response body will communicate the error occurred:

```
{
    "Error": "This string will report the error occurred"
}
```

DELETE `/verifier`

Delete an attestation technology registered in the TM.

Request JSON object

- `att_tech` (string): The name of the attestation technology.
- `inf_id` (int): The identifier of the infrastructure.

Example request:

```
{
    "att_tech": "keylime_v6_3_2",
    "inf_id": 1
}
```

Response JSON object

If the delete is completed successfully the TM returns `200` with a response body like the following:

```
{
    "Message": "verfier keylime_v6_3_2 successfully deleted"
}
```

In case of failure the returned code depends on the cause of the failure and the response body will communicate the error occurred:

```
{
    "Error": "This string will report the error occurred"
}
```

GET `/whitelist?whitelist_uuid=<whitelist_uuid>`

Get the information about a whitelist stored in the TM.

Response JSON object

- `_id` (int): The identifier of the whitelist (`whitelist_uuid`);
- `metadata` (json): The JSON object containing all metadata about the whitelist;
- `whitelist` (json): The JSON object which contains the whitelist.

Example response:

```
{
    "_id":2,
    "metadata":{
        "att_tech":"keylime_v6_3_2",
        "hash_algo":"sha265"
    },
    "whitelist":{
        "a_list_data":[
            "38ad97...2489b /pause",
            "1edf98...6d6b7 /usr/bin/local-path-provisioner"
        ]
    }
}
```

POST `/whitelist`

Insert a new whitelist in the TM.

Request JSON object

- `_id` (int): The identifier of the whitelist (`whitelist_uuid`);
- `metadata` (json): The JSON object containing all metadata about the whitelist;
- `whitelist` (json): The JSON object which contains the whitelist.

Example request:

```
{
    "_id":2,
    "metadata":{
        "att_tech":"keylime_v6_3_2",
        "hash_algo":"sha265"
    },
    "whitelist":{
        "a_list_data":[
            "38ad97...2489b /pause",
            "1edf98...6d6b7 /usr/bin/local-path-provisioner"
```

```
        ]
    }
}
```

Response JSON object

If the insertion is completed successfully the TM returns `200` with a response body like the following:

```
{
    "Message": "whitelist 2 added successfully""
}
```

In case of failure the returned code depends on the cause of the failure and the response body will communicate the error occurred:

```
{
    "Error": "This string will report the error occurred"
}
```

DELETE `/whitelist`

Delete a whitelist stored in the TM.

Request JSON object

- `_id` (int): The identifier of the whitelist (`whitelist_uuid`).

Example request:

```
{
    "_id": 2
}
```

Response JSON object

If the delete is completed successfully the TM returns `200` with a response body like the following:

```
{
    "Message": "whitelist 2 successfully deleted"
}
```

In case of failure the returned code depends on the cause of the failure and the response body will communicate the error occurred:

```
{
    "Error": "This string will report the error occurred"
}
```

POST `/attest_entity`

Start the remote attestation process for a specific entity.

Request JSON object

- `entity_uuid` (int): The identifier of the entity for which start the remote attestation process.

Example request:

```
{
    "entity_uuid": 2
}
```

Response JSON object

If the remote attestation process starts successfully the TM returns 200 with a response body like the following:

```
{
    "Message": "entity 2 - attestation thread started successfully"
}
```

In case of failure the returned code depends on the cause of the failure and the response body will communicate the error occurred:

```
{
    "Error": "This string will report the error occurred"
}
```

DELETE /attest_entity

Stop the remote attestation process for a specific entity.

Request JSON object

- `entity_uuid` (int): The identifier of the entity for which start the remote attestation process.

Example request:

```
{
    "entity_uuid": 2
}
```

Response JSON object

If the remote attestation process stops successfully the TM returns 200 with a response body like the following:

```
{
    "message": "attestation stopped successfully"
}
```

In case of failure the returned code depends on the cause of the failure and the response body will communicate the error occurred:

```
{
    "Error": "This string will report the error occurred"
}
```

GET /policy?entity_uuid=<entity_uuid>

Get the information about a whitelist stored in the TM.

Response JSON object

- `entity_uuid` (int): The identifier of the entity for which is defined the policy;
- `policy` (string): The value of the policy.

Example response:

```
{
    "entity_uuid": 1,
    "policy": "policy value"
}
```

POST /policy

Insert a new policy in the TM.

Request JSON object

- `entity_uuid` (int): The identifier of the entity for which starts the remote attestation process;
- `policy` (string): The value of the policy.

Example request:

```
{
    "entity_uuid": 1,
    "policy": "policy value"
}
```

Response JSON object

If the insertion is completed successfully the TM returns `200` with a response body like the following:

```
{
    "Message": "policy for entity 1 added successfully"
}
```

In case of failure the returned code depends on the cause of the failure and the response body will communicate the error occurred:

```
{
    "Error": "This string will report the error occurred"
}
```

DELETE /policy

Delete a policy from the TM.

Request JSON object

- `entity_uuid` (int): The identifier of the entity for which start the remote attestation process.

Example request:

```
{
    "entity_uuid": 1
}
```

Response JSON object

If the delete is completed successfully the TM returns 200 with a response body like the following:

```
{
    "message": "policy for entity 1 successfully deleted"
}
```

In case of failure the returned code depends on the cause of the failure and the response body will communicate the error occurred:

```
{
    "Error": "This string will report the error occurred"
}
```

GET /report

Get reports of an entity from the TM.

Request JSON object

- `entity_uuid` (int): The identifier of the entity for which starts the remote attestation process;
- `from` (string): This is an **optional** value and it represents the first date, in ISOFormat (`"yyyy-mm-ddThh:mm:ss"`), from which retrieving reports;
- `to` (string): This is an **optional** value and it represents the last date, in ISOFormat (`"yyyy-mm-ddThh:mm:ss"`), from which retrieving reports.

Example request:

```
{
    "entity_uuid": 1,
    "from": "2022-08-29T08:26:00",
    "to": "2022-08-29T08:35:00"
}
```

Response JSON object

If the request is served successfully the TM returns 200 with a response body like the following:

```
{
    "report_list":
    [
        {
            "_id": "62ea9ff6cd03ce4ce1b0aebd",
            "entity_uuid": 8, "trust": False,
            "state": [
                {
                    "att_tech": "keylime_v6_3_2",
                    "trust": False
                }
            ],
            "time": "Mon, 29 Aug 2022 08:26:42 GMT"
        },
        {
            "_id": "62eaa082be86d83bee075ff7",
            "entity_uuid": 8,
```

```
            "trust": True,
            "state": [
                {
                    "att_tech": "keylime_v6_3_2",
                    "trust": True
                }
            ],
            "time": "Mon, 29 Aug 2022 08:33:53 GMT"
        },
        ...
    ]
}
```

In case of failure the returned code depends on the cause of the failure and the response body will communicate the error occurred:

```
{
    "Error": "This string will report the error occurred"
}
```

GET `/status`

Get information about the current state of the TM.

Response JSON object

If the request is served successfully the TM returns `200` with a response body like the following, showing the adapters correctly loaded and working and the list of attestation processes running:

```
{
    "adapters_loaded":[
        "keylime_v6_3_2"
    ],
    "att_processes":[
        {
            "entity_uuid":1,
            "external_id":"90e71d86-13e0-4bd3-9ec4-1521f10a5194",
            "name":"attester",
            "att_tech":[
                "keylime_v6_3_2"
            ]
        },
        {
            "entity_uuid":8,
            "external_id":"d432fbb3-d2f1-4a97-9ef7-75bd81c00000",
            "name":"attester_2",
            "att_tech":[
                "keylime_v6_3_2"
            ]
        }
    ]
}
```