POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Verification of Software Integrity in Distributed Systems

Supervisors
Prof. Antonio Lioy
Dr. Ignazio Pedone

Candidate
Silvia Sisinni

Academic Year 2020-2021

To my family, who always supported and encouraged me

Summary

The last few years have seen a growth in popularity of Cloud Computing, a computational paradigm to deploy IT services, through which cloud providers make computing resources available at the request of users, ensuring greater flexibility, availability and cost reduction, without the need for the user to purchase and manage them. Cloud Computing achieves these benefits taking advantage of virtualization technologies, which can host services in a full virtualization environments, such as the Xen hypervisor or Kernel-based Virtual Machine, or in a lightweight virtualization environment, such as Docker. Nowadays the trend is towards the use of lightweight virtual machines, also named containers, considered by companies more advantageous for their flexibility, simplified deployment, compatibility with various operative systems, rapid availability, fine-grained subdivision of computational resources in micro-services. The Cloud Computing paradigm, however, while providing great benefits to users, introduced an entire whole world of security threats, such as isolation failure, economic denial of service, malicious insider, which expose companies and users to great security and privacy risks. Being able to verify the integrity and correct configuration of the software running on the cloud nodes is crucial to early detection of any type of tampering and breach, in order to react promptly to attacks. Remote Attestation is the process by which an external entity can assess the level of trustworthiness of a computational node; it works well for physical nodes, but it is not yet a well established process for virtual nodes, as traditional and lightweight virtual machines. Proposals have been published in the scientific literature but none of them completely satisfies the desirable requirements of scalability, low latency and availability in any deployment scenario. This thesis proposes a new solution to carry out periodic remote attestation of lightweight virtual machines deployed in a container runtime engine among those most used in current cloud platforms, such as Docker and containerd. The solution complies with Trusted Computing Group's specifications, relying on Trusted Platform Module 2.0, "Integrity Measurement Architecture" Linux's security module and Keylime as remote attestation framework. As proved by performance tests performed in laboratory, the proposed solution is highly scalable, adapts to different containerization technologies and guarantees low attestation latency as the number of containers, deployed on the cloud node, increases.

Acknowledgements

I cordially thank Prof. Antonio Lioy for entrusting me with this work, which has given me the opportunity to enrich my professional knowledge.

I also sincerely thank Dr. Ignazio Pedone for guiding and supporting me with his precious advice in the realization of this work.

Contents

Chapter 1

Introduction

Over the years, ICT infrastructures have developed and evolved, passing from a centralized scheme, where applications and data reside on a single processing node, to a scheme in which numerous distributed components contribute to the storage and processing of data. Among distributed systems, the most widespread currently is *Cloud Computing* which changed the delivery model for IT services based on Internet, providing users with dynamically scalable and virtualized computing resources. Cloud Computing is based on the idea of delegating the management and delivery of software and hardware resources to third-party companies (cloud providers) which, being specialized in that particular field, can offer a better quality of service at lower costs. Thanks to this paradigm, enterprises can achieve significant cost savings by purchasing IT resources based on their actual demands, according to the "pay-per-use" model, at the same time leveraging leading-edge technologies for their computational and storage needs. For these reasons cloud computing, initially aimed only at large companies, has grown rapidly, spreading to small and medium-sized businesses and currently also to consumers.

Nowadays in the world there is a myriad of *Internet of Things* (IoT) devices which allow companies and individuals to take advantage of a great variety of sophisticated and targeted services. All of these devices produce a huge amount of data that needs to be processed and stored. Managing this new scenario only with the Cloud Computing paradigm would lead to an overload of the backbone network of Internet, a large overhead in the Cloud servers and would not allow the creation of fully real-time services as IoT requires. These new needs are met by *Edge Computing* and *Fog Computing*, two new distributed paradigms currently emerging, which differ in design and purpose and both play a complementary role to that of Cloud Computing. By allowing for the processing and storage of part of the data locally, they achieve a smooth transition to fully decentralized systems. Edge Computing is a distributed paradigm where computation and storage usually occur either directly on the device that generates data or on a gateway device that is physically close to sensors. Each edge device can act as a server in the edge network, so while cloud computing is typically distributed to dozens of servers, edge computing is distributed over hundreds or thousands of local nodes. Fog Computing, instead, moves part of edge activities to fog nodes more distanct from sensors and actuators, but situated within the same LAN, and it brings more intelligent data analyses to servers in the Cloud; so, it bridges Edge Computing and Cloud Computing. The local processing and storage, even if partial, of the enormous amount of data generated by these devices bring great advantages such as low network traffic, real-time data analysis, low latency, reduced operating costs. Fog and Edge computing are designed to adapt to these scenarios and certainly in the near future more and more companies and service providers will want to use and benefit from them.

If on the one hand distributed infrastructures offer enormous benefits to companies and consumers, on the other hand they introduced new criticalities about security and privacy. Nowadays, attacks by cybercrime are unrelenting and continuously cause very serious damages to companies, government organizations and individual users. Perimeter-focused security architectures are no longer suitable to protect new ICT infrastructures, which are characterized by a dissolution of traditional boundaries. Modern security paradigms focus on the concept of "trustworthiness" of a computational entity, which concerns the determination of its behavior: it is *trusted* if it

behaves as expected for the intended purpose, consequently it is *trustworthy* if its behavior is predictable. The determination of the trustworthiness level of a computational node requires the periodic verification that all its software and configuration files have not been tampered with by malicious attackers; this can be achieved through *Trusted Computing* (TC) technology, which has the *Trusted Platform Module* (TPM) chip as its founding element. The TPM is a secure cryptoprocessor whose primary scope is to protect the integrity measurements, which are digests calculated through cryptographic hash algorithms on the platform components, inside its special memory locations called *Platform Configuration Registers* (PCRs), and to report them to a third party in an authenticated way. These TPM capabilities, together with the platform components that perform the measurements, enable the *Remote Attestation* (RA) process through which a platform, the *Attester*, demonstrates its integrity state to an external entity, the *Verifier*. The integrity verification can concern only the boot phase of a system or also include its runtime. *Integrity Measurement Architecture* (IMA) is the Linux kernel module responsible for measuring the files accessed by the system at runtime; each measurement acquired by IMA is extended in a specific PCR and stored in a log file, whose format is established by the configured IMA template.

Modern infrastructures heavily rely on virtualization techniques in order to optimize the use of IT resources, so user services are often hosted not directly in physical machines but in *Virtual Machines* (VMs) managed by the *Hypervisor*, a software layer, installed on a physical server, that controls and emulates the hardware resources, assigning them dynamically to the VMs. Since VMs are widely used in cloud environments, extending the RA process to them is extremely important. However, while remote attestation of physical nodes is a well-established process, attesting VMs is still a challenging task because VMs do not always have direct access to the TPM and, even when this is available, the number of available PCRs is insufficient for managing all the VMs running on a physical machine.

Further challenges have arisen in recent years as *Lightweight Virtualization* techniques began to be adopted in cloud environments, replacing classic virtualization in some use cases. *Containers*, which are the computational entities provided by the lightweight virtualization, have near bear-metal performance, unlike traditional VMs which entail a considerable overhead of management; since they offer an extremely advantageous price/performance ratio, companies are encouraged to adopt them for deploying their services. While VMs have their own operating system and the isolation among them is guaranteed by the hypervisor, containers share the kernel of the host system and isolation among them is achieved through kernel features; hence, if the host environment is not properly configured, they are more exposed to the lack of logical isolation and maliciuos insiders than VMs. It follows that the ability to perform the RA process of containers and their underlying host system is fundamental to guarantee the trustworthiness of containerized services.

Some solutions have been proposed and discussed for realizing RA of containers, such as *Docker Integrity Verification Engine* (DIVE) [1] and *Container-IMA* [2]. Both solutions allow to perform a separate evaluation on the integrity state of each container running on the platform; however, they have some limitations and their implementations rely on TPM 1.2, a specification deprecated and substitued by TPM 2.0. DIVE requires the Docker engine to be configured with a specific storage driver, Device Mapper, not allowing to use other storage drivers which, in some situations, could have better performance. Moreover, DIVE identifies the measurements relating to a specific container through the virtual device identifier that Device Mapper assigns to all processes running in that container; however, when a container is terminated and a new one is created, the new container acquires the same virtual device identifier assigned to the previous container, inheriting all the measurements generated by the terminated container and causing, in this way, an integrity failure when its status is evaluated by a Verifier. Container-IMA is particularly effective for ensuring the privacy of container measurements in multi-tenant environments, because a Verifier receives only the measurements related to the container that it has to attest. However this solution is based on the assumption that there is no need to attest to the entire host system to assert that the environment, in which the container is running, is trusted. Furthermore, the privacy guarantee of the measures concerning a container is based on a shared secret between the kernel space of the system where the container is running and the Verifier, but the mechanism by which the secret can be shared in a secure way is not well defined.

Purpose of this work was to overcome some criticalities of the previous solutions and provide

an implementation of container attestation which relies on TPM 2.0 specification. The thesis work consisted in developing a new IMA template that allows to identify the measurements related to a specific container directly through its container ID; this was realized by exploiting a behaviour of containarization technologies, which assign control groups with name equal to the container full-ID to the processes running in a specific container. Furthermore, the thesis work also concerned the extension of Keylime, a remote attestation framework able to attest physical nodes by relying on the TPM 2.0 specification. The Keylime code was appropriately modified in order to make the Keylime Verifier able to interpret the new IMA template and support attestation of individual containers. Moreover, it was decided to exploit the algorithm agility introduced by TPM 2.0, allowing to use hash algorithms with a higher security level than SHA-1 during the integrity verification of the IMA measurements. Then, in order to optimize the attestation times, it was decided to check, at each attestation cycle, only the last IMA measurements not yet attested, instead of the entire measuements log. In order to support the registration of containers inside the Keylime Verifier, new REST APIs were also added to the Keylime framework and some already present were adapted.

The Keylime framework, extended for container attestation, was then integrated with the Trust Monitor, a monitoring entity developed by the TORSEC research group for verifying the integrity state of all the components of a *Network Functions Virtualization* (NFV) platform. In modern networks, in fact, many functions are no longer performed by dedicated hardware appliances, but are deployed inside VMs or containers running on general purpose computational nodes. Consequently, all these virtual components must be constantly monitored to verify their integrity through the RA process, analogously to what has to be done for virtualized components in a cloud infrastructure. The thesis work provided the Trust Monitor with a new tool which takes advantage of the TPM's capabilities introduced by the 2.0 specification for attesting containers deployed in a NFV infrastructure.

Chapter 2

Trusted Computing and TPM 2.0

This chapter provides an overview of *Trusted Computing* (TC), examining the reasons that led to its definition and to the specification of the *Trusted Platform Module* (TPM), and what are the functional capabilities offered by TPM 2.0, pointing out the introduced innovations compared to TPM 1.2.

2.1 Trusted Computing and TCG

The problem about computer security was highlighted since 1960's by the United States military. They saw in the use of resource-sharing systems, emerging during those years, an advantage for the increase of computer performance and efficiency, but also a threat and a security risk since users had the possibility to read each other's data. For this reason, in 1967 U.S.' *National Security Agency* (NSA) promoted several research projects which were the basis for computer security. In 1981 the Department of Defense (DoD) founded the Computer Security Initiative (CSI) which, at the "IEEE Symposium on Security and Privacy", presented a paper [3] containing a first definition of Trusted Computing systems. These were described as systems that "employ sufficient hardware and software integrity measures to allow its use in processing multiple levels of classified or sensitive information". In December 1985, the DoD published the "Trusted Computer System Evaluation Criteria" (TCSEC), commonly known as the *Orange Book*, a standard for estimating the effectiveness of computer security controls built into automatic data processing systems. This document provides the first formal definition of the *Trusted Computing Base* (TCB) of a computer system, stated as the set of all the elements of the system responsible for supporting the security policy and the isolation of the objects (code and data) used to protect the system. The TCB, whose boundaries constitute the "security perimeter", includes hardware, firmware and software critical to protection and must be designed and implemented such that, even if system components out of it become untrusted, this does not compromise system protection [4]. The criteria expressed in the Orange Book were ahead of the times, because they were introduced in the era of mainframe systems, when the computer security was intended mostly as physical security of the system, with few logical security delegated to the operating system, so the "science and discipline of trusted computing" was neglected for a long time [5].

In the 1990s, with the widespread of Personal Computers (PCs), connected each other through Internet, the computer industry understood that the security in PCs had to be enhanced in order to allow the diffusion of important applications, sensitive in terms of security and privacy, as the electronic commerce. In fact, PCs were primarily designed having in mind ease of use, with little thought to their security. IT vendors, however, tried to improve computer security, producing software to protect the system, like firewalls and Intrusion Detection Systems (IDS), to discover viruses, worms, trojans and so on, but their solutions were single vendor focused and not interoperable. Moreover, it became clear that all attempts to detect malicious code through software-only solutions were ultimately circumvented, while it was quite easy to detect software compromise with a little bit of hardware support [6].

This approach was fostered by the Trusted Computing Platform Alliance (TCPA), formed in 1999 by Microsoft, Intel, IBM, Hewlett Packard (HP) and Compaq with the aim to promote the development of Trusted Computing by relying on both hardware and software implementations. The TCPA proposed a hardware anchor for PC's security, called *Trusted Platform Module* (TPM), on which secure systems could be built. Starting from August 2000 many releases of the TPM specifications were published, till the publication of the TPM version 1.1b on 22 February 2002. In order to keep low the chip cost, the designers developed a minimal chip intended to be physically affixed to the motherboard of a PC, with a command set containing only the needful security functions and moving to the software layer all the functionalities for which an hardware implementation was not necessary. IBM was the first that integrated the TPM 1.1b into its PCs, soon HP and Dell followed it and, by 2005, almost all commercial PCs and servers had one attached to their motherboard, or had the slot to add it afterwards.

The TPM 1.1b specifications contained the following basic functions [7]:

- key generation (limited to RSA keys);
- storage of integrity metrics in special registers called *Platform Configuration Registers* (PCRs);
- reporting of integrity metrics;
- secure authorization;
- use of *Attestation Identity Keys* (AIKs) associated to TPM identity;
- cryptographic operations.

A new network entity called *privacy Certificate Authority* (CA) was designed, whose role was to prove that an AIK key generated in the TPM came from a real TPM without identifying it, so guaranteeing the privacy. The protection of the TPM from physical attacks was out of the scope of the specification and was left as an area where vendors could differentiate their chips, while any software attacks were considered within the scope of TPM-based security.

In 2003, on the initiative of AMD, Hewlett Packard, IBM, Intel, Microsoft, Sony, Sun Microsystems and other companies (for a total of fourteen members, to which many other companies joined over the following years), the TCPA became the *Trusted Computing Group* (TCG), which inherited TPM version 1.1b and published in 2004 the version 1.2 of the specification [8], which was developed with the aim to overcome the drawbacks of TPM 1.1b. The main changes regarded [7]:

- a standard software interface, so that TPMs produced by different vendors would not required different drivers;
- a mostly standard package pinout;
- a protection against dictionary attacks for discovering the authorization passwords for using the keys protected in the TPM;
- the implementation of a new method for anonymizing keys, *Direct Anonymous Attestation* (DAA) and delegating key authorization and administrative functions;
- the introduction of a small nonvolatile RAM (usually about 2 kB) for storing the certificate of the TPM's *Endorsement Key* (EK);
- the definition of a kind of keys specifically intended for migration from a TPM to another, called *Certified Migratable Keys* (CMKs), in order to simplify the migration mechanism;
- the synchronization of a TPM internal timer with an external clock, in order to enable the possibility to add a timestamp to a sign.

TPM 1.2 was a success, it was deployed on most x86-based client PCs starting from 2005 and on most server starting from 2008, resulting in more than 1 billion TPMs 1.2 deployed in computer systems. However, just the presence of hardware is not valuable if there isn't software that exploits its capabilities, so Microsoft developed a TPM driver for Windows, IBM developed an open TPM driver for Linux and applications that used TPM began to be deployed.

While the TPM 1.2 began to be almost ubiquitous on all systems, the TCG started the work on the TPM 2.0 specification. The trigger was the publication, in 2005, of the first significant attack on the SHA-1 digest algorithm, which was heavily used in the TPM 1.2 architecture. Although the reliability of the TPM 1.2 was not compromised by this attack, the TCG decided to create a new specification that was *agile* with respect to digest algorithms, considering the common axiom in cryptography, according to which algorithms become weaker over time, never stronger [7]. The TCG designers chose to not hard-code any algorithm in the TPM 2.0 specification, but rather to incorporate an algorithm identifier that allowed to use a wide range of cryptographic algorithms in the TPM and to enlarge the set of algorithms over time without the need to change the specification. The *algorithm agility* was the first motivation for a new TPM specification, but during the work many other generalizations were added, such as [7][9]:

- the *Enhanced Authorization* (EA), which increased the flexibility of the authorization methods and at the same time it reduced the implementation cost, by:
  - unifing the mechanism that authorizes the use, delegates the use and migrates objects and entities in the TPM;
  - allowing authorization with clear-text passwords and Hash Message Authentication Code (HMAC);
  - allowing the construction of an arbitrarly complex authorization policy of an object through multiple authorization qualifiers;
- block symmetric key encryption, which removed the barrier on the size of TPM structures, while TPM 1.2 structures have to be compact enough for being encrypted with a 2048-bit RSA key in a single encryption operation;
- support for the *Elliptic Curve Cryptography* (ECC) algorithms;
- multiple key hierarchies to accommodate different user roles;
- dedicated BIOS support;
- simplified control model;
- a compilable specification, which has the advantage of being much less ambiguous since, in case of doubts, it permits to be compiled by an authoritative emulator to see how it should work.

2.2 Trusted Platforms

According to the TCG specifications, something is *trust* if it conveys an expectation of behavior. It is worth notice that predictable behavior does not mean correct behavior. This definition implies that a platform is trusted if it behaves as expected for a specific purpose, from which it follows that, in order to establish if a platform behaves as expected, it is important to determine its identity, that is to determine the identity of its hardware and software components. The TPM is the component proposed by TCG for collecting and reporting these identities in a way that permits to determine the expected behavior and, from that expectation, to establish trust [9]. The TPM is a system component whose state is separate from the host system on which it reports and interacts with the host system only through the interface defined in the specification.

The TCG maintains the definition of Trusted Computing Base (TCB) stated in the Orange Book. In fact, in the TPM specifications a TCB is defined as the collection of system resources (hardware and software) that have the responsability to enforce and maintain the security policy

of the system. An important property of a TCB is that it cannot be compromised by any hardware or software component that is not part of the TCB. The TPM is not the TCB of a system, but it is a component that allows an external entity to verify if the TCB has been compromised. The TPM can also be used to prevent the system from starting in the case the TCB cannot be properly instantiated [9].

2.2.1 Roots of Trust (RoTs)

The TCG defines the *Roots of Trust (RoTs)* as the minimum set of system elements on which the trustworthiness of the platform is based and whose misbehavior is not detectable. A TPM can accomplish its design goals if and only if the RoTs are properly implemented. A component or a collection of components required to instantiate a RoT is called *Trusted Building Block (TBB)*. Even if it is not possible to check at runtime if the behaviour of the RoTs is correct, there are certificates that provide assurances that they have been implemented in a trustworthy way. For example, a certificate provided by an independent testing lab may report the Evaluated Assurance Level (EAL) of a TPM, consequently providing confidence in the correct implementation of its RoTs. In addition, a platform manufacturer certificate may provide assurance that the TPM was properly attached to the matherboard of a machine compliant with the TCG specifications, so that the RoTs may be deemed trusted. For example, most designs of TPM 1.1b were both FIPS 140-2 Level 1 and Common Criteria (CC) EAL3 certified, while most TPM 1.2 and TPM 2.0 designs are validated based on FIPS 140-2 Level 2 and CC EAL4+. Tests are conducted according to a specific TCG protection profile.

The TCG requires that a *Trusted Platform (TP)* provides at minimum the following three RoTs:

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

Root of Trust for Storage (RTS)

The RTS is the TPM memory, which has the property to be shielded from access by entities other than the TPM [9]. Since this feature needs to be trusted, the TPM acts as a RTS.

The TPM memory locations (called *Shielded Locations*) can contain:

- non-sensitive information (for example the digest contained in the PCRs) which does not need to be protected from disclosure, so the access for reading is never denied, while the access for writing follows a specific policy;
- sensitive information (for example the private part of an asymmetric key) to which the TPM denies access without proper authorization.

Root of Trust for Measurement (RTM)

The RTM supports the integrity measurement of the TP by calculating digests taken on configuration data and program code, and sending them to the RTS. The concept at the base of the integrity measurement of a platform is the *Transitive Trust*, by which the trust in a software component is used to evaluate the trustworthiness of the subsequent software component which will take the control of the platform. The TPM cannot implement the RTM, since it is conceived as a slave device that receives commands. Instead, the RTM of a platform is the Core Root of Trust for Measurement (CRTM) when it is executed by the CPU at system reset. Typically, the CRTM is a small subset of the BIOS, the first set of instructions that get the control of the system; its purpose is to record in the TPM which BIOS is being used to boot the system before passing control to the full BIOS. CRTM is the starting point of a chain of trust that is transferred to the subsequent software components [9] [6].

Root of Trust for Reporting (RTR)

The RTR reports on the contents of the RTS. Typically an RTR report is a digitally signed digest calculated on the values of some Shielded Locations within a TPM, such as:

- the content of PCRs, which provide evidence of the platform status; in this case, the RTR report is called *Integrity Report*;
- audit logs;
- key properties.

The reports cannot be created on Shielded Locations that contain sensitive information, such as the private part of asymmetric keys and authorization passwords. The TPM can implement the RTR since it has the cryptographic capabilities to create an RTR report.

The interaction between the RTR and RTS is a critical point of the TPM design, since its misbehaviour would prevent to create an accurate RTR report. TCG recommends that RTR and RTS implementation:

- resists to all kinds of software attacks and to the physical attacks specified by the TPM's Protection Profile;
- provides an accurate digest of the integrity metrics that took place in the platform.

Over the year, flaws were discovered in the RTR and RTS implementation of some TPMs certified EAL4+, showing how critical is a proper implementation of the RoTs.

2.2.2 Secure Identity

The TPM reports on the integrity state of a Trusted Platform by quoting the PCR values. For this quote to be used by external entities to check the platform state, it is essential to:

- identify the RTR (hence the TPM) that issued the quote;
- have a proof of the proper binding between that RTR and the RTM that took the measurements.

The RTR (and TPM) identification is accomplished by means of non-migratable asymmetric keys called *Endorsement Keys* (EKs), derived from an *endorsement seed* contained in the TPM. The seed is statistically unique for the TPM; from this it follows that the probability to have two TPMs with the same EK should be insignificant and all the EKs generated from the same seed represent the same TPM and RTR. Moreover, the fact that EKs are non-migratable ensures that they exist for a given TPM and never outside that TPM. The nominal method of establishing trust in a key is through a certificate that attests the authenticity of the key. The TPM manufacturer can generate an EK from the endorsement seed and provide a certificate of authenticity for that EK, commonly called *Endorsement Certificate*. However, the EK does not need to be permanently installed in the TPM, so the TPM owner may use the endorsement seed and recreate the EK for which he provides another certificate of authenticity [9]. Instead, the proof of the physical binding between the RTM and the RTR can be provided through a *Platform Certificate* emitted by a Certifying Authority.

EKs can raise a privacy problem because, since they represent the TPM identity, a direct use of them could permit the creation of activity logs, which could reveal personal information that the user of a platform would not otherwise want to reveal to entities that aggregate data [9]. To counter this issue, TCG recommends not to use EKs either for signing or for encrypting, but only to decrypt certificates of other non-migratable keys generated by the TPM (the *Attestation Keys* (AKs) or AIKs) during the *Attestation Key Identity Certification* protocol. This happens at request of the TPM owner with the cooperation of an *Attestation CA* (or *Privacy CA*) and

has the purpose to certify that a given AK has been generated by a valid TPM. AKs, in turn, can only be used to sign a digest that the TPM has generated, like the digest calculated on the content of PCRs (the *quote* operation). In order to prevent forgery, an AK cannot be used to sign digests generated out of the TPM, because they could be related to data that appears to be authentic and TPM-produced but is not. In particular, the data on which the digest is calculated can be produced inside or outside the TPM but, for data coming from outside, the TPM checks that the first bytes are not equal to TPM_GENERATED_VALUE before generating the digest and signing it with the AK. In this way, an AK can be used to sign values that reflect the TPM state or for general signing purposes [9]. Although the Attestation CA could keep track of the 1:1 association between AKs and EKs, the end user has the possibility to choose a CA that promises not to. Once the AK certificate is created, it does not have any link back to the EK and the AK can be used to identify the TPM. Moreover, if the AK authorization is known only to one user, it can be used also to identify that particular user of the TPM.

After the TPM 1.1b specification, however, few Attestation CAs were been implemented. This was probably due to a dearth of EK certifications provided by the TPM manufacturer because, if the EK is not certified by a trusted entity, its trust and privacy properties are equivalent to any other asymmetric key generated by pure software methods. Therefore, by itself, the public portion of the EK is not privacy sensitive. However, this lack of Attestation CA implementations caused strong skepticism in the privacy community, as this could lead full identification of all attestations. These considerations led to the inclusion in the TPM 1.2 specification of a new method for authenticating AKs, called *Direct Anonymous Authentication*.

Direct Anonymous Attestation (DAA)

DAA is a cryptographic protocol based on the group signatures mechanism. A group signature gives to a group of people the possibility to sign messages as "members of the group", without having to share a common secret, which exposes to a Break Once Run Everywhere (BORE) attack, and without exposing the identity of the member of the group that signed a particular hash [6]. Nevertheless, if the individual secret owned by a member of the group is revealed, the secrets owned by the other members of the group are not compromised, so it is possible to invalidate only the revealed secret.

DAA provides a means to certify an AK as member of the group of TPM AKs, without the need for each AK to be individually certified by a CA. A DAA Authority authenticates some TPM secrets as belonging to an authentic TPM and this authentication is done once, before the TPM is shipped out to its final owner or afterwards, when the TPM is already in field.

The DAA protocol is composed of the following phases:

1. the DAA Authority issues a DAA certificate, after it verified that the TPM belongs to the group of authentic TPMs;
2. the TPM uses the DAA key to sign a message and a verifier uses the DAA certificate to validate the signature, without having the possibility to know the identity of the TPM or even recognize that multiple signatures derive from the same TPM;
3. the DAA Authority can revoke a DAA certificate; this happens if the TPM chip becomes compromised, for example its private keys become exposed.

2.2.3 Attestation Hierarchy

Integrity evaluation of Trusted Platforms is based on an Attestation Hierarchy as described in figure 2.1.

1. The *Endorsement Certificate*, issued by an external entity (typically the TPM manufacturer), attests that the TPM (which holds the RTS and the RTR) is authentic and compliant with the TCG specifications. In particular, the certificate vauches that a given EK has been generated by a genuine TPM and can be used to identify it during an attestation process.

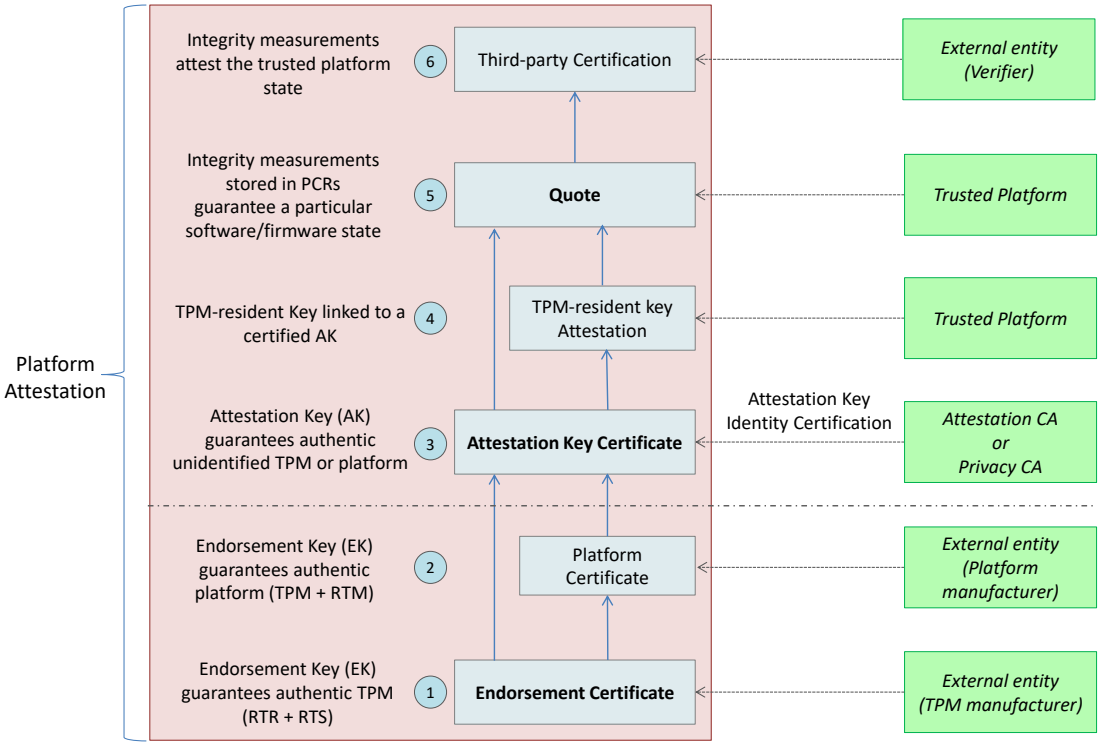

Figure 2.1. Attestation Hierarchy

2. The *Platform Certificate*, issued by an external entity (typically the platform manufacturer), attests that a given platform implements all the RoTs needful to consider it a Trusted Platform and that they have been correctly integrated through trusted paths. In particular, this certificate vouches that a given EK has been generated by a genuine TPM, which is integrated with the RTM through a trusted path.

3. The *Attestation Key Certificate*, issued by an Attestation CA (or Privacy CA) at the end of the *Attestation Key Identity Certification* procedure, attests that an AK key has been generated by an authentic but unidentified TPM, is a non-migratable key and can be used to sign the contents of Shielded Locations. The Attestation CA typically provides this kind of attestations relying on attestations of type 1 and 2.

4. A certified AK can be used to certify that other AKs are resident in the same TPM and have the same properties as the certified AK. This kind of attestation, issued by the Trusted Platform, is a sign performed with the private part of a certified AK key over the digest calculated on the properties of the key to be certified. Attestation of type 4 relies upon attestation of type 3 for the sign verification.

5. The *Quote*, issued by the Trusted Platform, attests a particular software/firmware state in the platform. A quote is a signature performed with the private part of a certified AK over the digest calculated on the measures of the software/firmware stored in the PCRs. This kind of attestation relies upon attestation of type 3 or 4 for the sign verification.

6. The *Third-party Certification* is emitted by an external entity (*Verifier*) and attests the measurements of a Trusted Platform in order to vouch a particular state of its software/-firmware. It's a credential containing the measurements of the software/firmware together with the state they represent.

2.2.4 Trusted Boot

For determining if a platform has a trusted status, it's necessary to evaluate the trustworthiness of all the software components that take the control of the platform, starting from the boot sequence till the runtime. *Transitive trust* is the concept on which this evaluation is based and has its foundation in the RoTs. It uses the trustiness in the RoTs as the entry point of a chain of trust, which is transferred in a transitive way from an executable function to the next one that takes the control over the machine.

Transitive trust may be realized in two different ways, both supported by the TPM:

1. evaluating if a function is trusted before passing the control to it, repeating the same procedure in the subsequent functions;
2. measuring a function, storing that measure in the RTS and passing the control to it anyway; the same procedure is repeated in the subsequent functions, leaving to an independent entity to evaluate the platform status in a transitive way, so that if a function is deemed untrusted, the measures it took are not reliable, hence all the subsequent functions are considered untrusted as well.

The "measure" is a digest calculated with a cryptographic hash function on anything of meaning to evaluate the trusted state of a platform, such as code, data values or an indication of the signer of some code or data [9]. As already mentioned, this digest is stored in a special shielded location of the RTS in the TPM, the PCR, whose value can only be changed through the operations:

- *Reset*, that sets the PCR value to all-zero and is performed at power-on of the platform; it can occur afterwards only if the PCR has an attribute that allows it to be reset (typically PCRs with index greater than 15);
- *Extend*, that lets to store an accumulative hash in a PCR; it takes the current value of the PCR, concatenates an input value, calculates a digest with a cryptographic hash function on the resulting concatenation and stores the output of this operation as the new PCR value:

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

It is possible for a single PCR to store all digests of the boot sequence, since the *extend* operation allows to accumulate an indefinite number of measurements in a PCR. However, in order to simplify the evaluation of the stages of the platform from the boot to the runtime, normally multiple PCRs are provided in a TPM, each one dedicated to store measurements of different modules (the BIOS, the OS boot loader, ...). All PCRs that are extended with the same hash algorithm constitute a *PCR bank*. While the TPM specifications 1.1b and 1.2 forseen only one PCR bank for the SHA-1 algorithm, the TPM 2.0 specification allows the presence of multiple banks of PCRs, at least the SHA-1 bank and the SHA-256 bank.

The digest contained in a PCR is statistically unique and, if the sequence of the extensions performed in the PCR is known, it is possible to know the trusted value that a particular sequence of measurements should have. If the sequence of the extensions is not predictable, the trusted values of the PCRs cannot be known a-priori. To handle this case, the RTM keeps *Measurement Logs (MLs)* in which each entry represents a *Measurement Event (ME)*, that is a change in the system state. The PCR values can be used to determine the integrity of the MLs, and an individual log entry can be used to determine if the event indicated by it caused a change in system state that is evaluated acceptable [9]. The RTM implementers decide what information a log entry has to contain to represent the Measurement Event. The process of attesting the integrity measurements stored in the PCRs is called *Integrity Reporting*. The concepts of integrity measurement, logging and reporting are the basis of a Trusted Platform and are motivated by the fact that the platform's software can be compromised at any time during its life-cycle. Thus it is necessary that the Trusted Platform has the capability to accurately report the states, including undesirable or insecure ones, in which it enters, so that an independent process may evaluate the state and take actions accordingly.

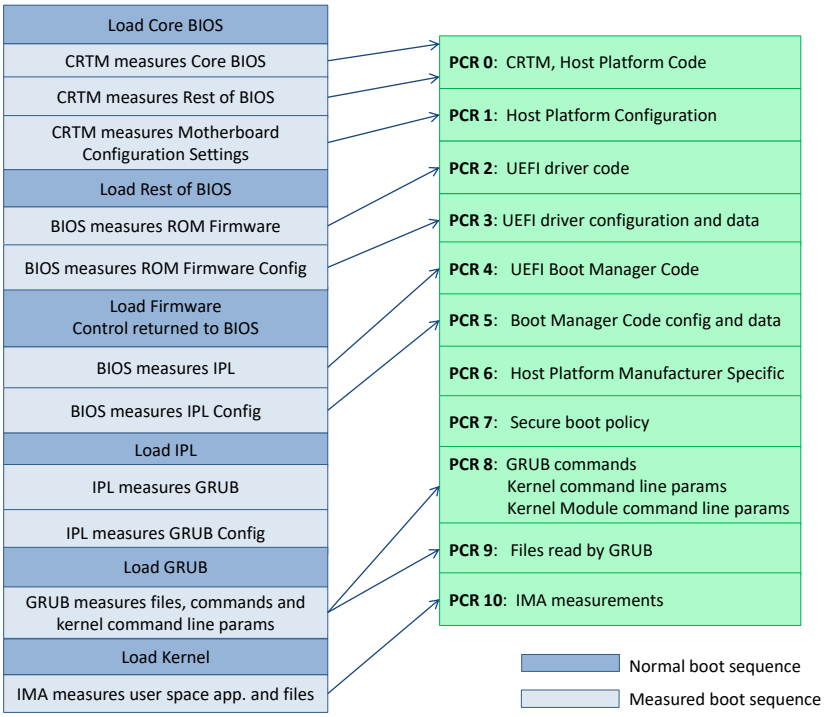

Figure 2.2. Trusted Boot (source: [6]).

The *Measured Boot* is a process whereby all the software components and configuration files involved in the system boot are measured and the digests stored in a specific PCR. Typically, PCRs with index less than 10 contain measures related to the system booting, those with index from 10 upwards contain measures of events happened after the kernel booted. The boot sequence depends on the particular platform configuration, but it mostly can be described as shown in figure 2.2. The Measured Boot starts from the CRTM (typically a subset of the BIOS) which firstly measures itself, the rest of the BIOS and extends the measurements in PCR 0, then it measures the motherboard configuration settings and extends the result in the PCR 1, finally it passes the control to the rest of the BIOS. This measures the ROM firmware (card's BIOSes) and extends the measure in the PCR 2, then measures the ROM Firmware Configuration and Data and extends this information in the PCR 3 before passing control to the ROM Firmware. After the ROM Firmware finishes the execution, the control returns to the BIOS, which measures the Initial Program Loader (IPL) code, that usually is the content of the Master Boot Record (MBR), referred to as the primary boot loader, and extends the measurement in the PCR 4, then measures the IPL Configuration and Data and extends the information in the PCR 5. PCR 6 contains information about the host platform manufacturer, PCR 7 regards the secure boot policy, the security state and the debug state. Then the control is passed to the IPL, whose purpose is to load the secondary boot loader, which for default is GRUB in the x86 platforms, and to pass the control to it. GRUB extends in the PCR 8 any grub command executed, any command line parameter passed to the kernel and the modules of the kernel, while it extends in the PCR 9 any file it reads and finally it passes the control to the kernel. This process establishes a chain of trust that goes from the CRTM to the kernel; since the chain of trust is created only once at platform reset, it is called a *Static RTM (S-RTM)*.

Some new processor architectures provide a different method to establish a new chain of trust without rebooting the platform. With this method, the CPU acts as the CRTM, applying protections to portions of the memory it measures [9]. Since the chain of trust implied by the RTM may be re-established dynamically, without the need of resetting the platform, this method is called *Dynamic RTM (D-RTM)*. The Intel processors equipped with the Trusted Execution Technology (TXT) are an example of implementation of D-RTM.

A Measured Boot can be of two different kinds:

- a *Secure Boot*, in which the measurements stored in the TPM are used to check if the selected subsequent component is trusted; this check can be done by verifying the digital signatures of the software components; in the case the signature is not valid, the control is not passed to the subsequent component and the system stops booting; in this way the implementer can enforce the machine to boot only into a trusted state by selecting a trusted firmware; Secure Boot could be implemented without the TPM as well;

- a *Trusted Boot*, in which the measurements recorded in the TPM are not used to prohibit booting in an insecure state, rather they are used to report the state of the platform to an independent entity, that can verify if the system booted in a secure way.

2.3 TPM

The TPM 1.2 specification was the TCG's first attempt to solve the security problems that the advent of Internet raised, particularly for the new applications that Internet enabled like electronic commerce. The main issues that TCG addressed were the following [7].

Identification of devices Before the TPM specification, device identification was performed with MAC or IP address, which are not security identifiers since they can be easily re-configured. TCG provided a way to prove device identity by means of TPM-generated non-migratable asymmetric keys.

Secure Generation of Keys The TCG forsaw for the TPM to have an internal *Random Number Generator (RNG)*, which allowed a more secure creation of keys with respect to previous solutions.

Secure storage of keys The TPM provides two ways to protect user's keys and data from software attacks. The first consists in storing them in the TPM's *Shielded Locations*, that can only be accessed through *Protected Capabilities*; in this way objects are protected from disclosure, tampering and deletion without authorization, like having them in a bank vault. The second technique consists in encrypting data with keys internal to the TPM and storing them in memory locations outside the TPM, called *Protected Locations*; data stored in this way are protected from disclosure but not from tampering and deletion, nevertheless this technique provides virtually unlimited amounts of secure storage.

NVRAM storage The reason that led TCG to add a Non-Volatile RAM (NVRAM) internal to TPM was the need to store the EK certificate inside the TPM. Storing the EK certificate in the hard disk of the platform was not a good solution because IT organizations, when they received a new device, often wiped the hard disk and installed their own software load, causing the EK certificate to be erased.

Integrity platform attestation When systems didn't have TPMs, their integrity was attested by leveraging software solutions. But, if software was compromised, it reported an healthy status of the system even when it was not so. The presence of the TPM in the platform allows to report the real state of the platform even when it has been compromised.

In the following discussion we will focus on the version 2.0 of the specification, pointing out its architecture, the main differences with the previous standard 1.2 and the various types of implementations.

2.3.1 TPM 2.0 Architecture

The functional units that compose the architecture of TPM 2.0 are represented in figure 2.3.

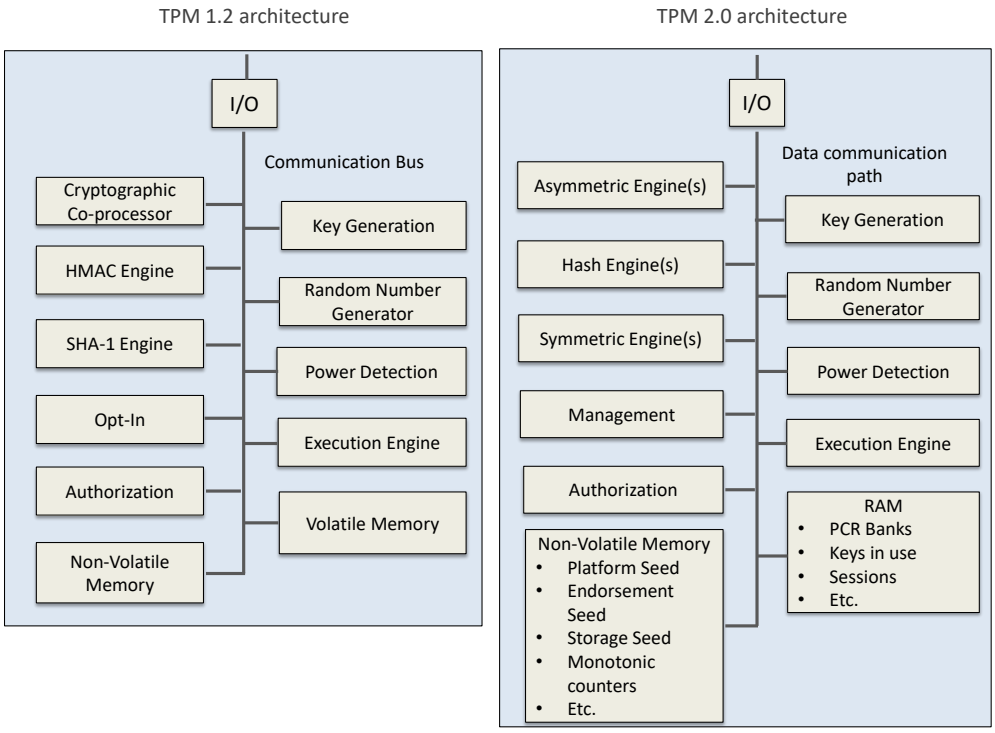

Figure 2.3. TPM 1.2 Architecture vs TPM 2.0 Architecture (source: [8] [9]).

I/O buffer

The I/O buffer is a memory area that permits the communication between the TPM and the host system. The Input buffer contains command data sent by the host system, while the Output buffer contains the response data produced by the TPM. TPM 2.0 specification doesn't require the I/O to be physically separated from the host system, it could be a shared memroy; however, when the command processing begins, command data has to be in a TPM shielded location.

Cryptography Subsystem

The Cryptography Subsystem contains the TPM's cryptographic functions that are invoked by the Execution Engine or the Authorization Subsystem. It is constituted by Asymmetric Engine, Hash Engine, Symmetric Engine, Key Generation, Random Number Generator. The TPM 2.0 specification contains both asymmetric (as 1.2 version) and symmetric algorithms, for encryption/decryption and signing/signature-verification operations. With regard to the algorithms recommended by TCG, the specification refers to another TCG document, the "TCG Algorithm Registry" [10], containing the list of algorithms to which the TCG has assigned an *algorithm identifier*. The algorithms are classified as:

- **S**, "TCG Standard";
- **A**, "Assigned";
- **L**, "TCG Legacy".

Currently, the only supported *asymmetric algorithms* are RSA (S) and prime field ECC (S), used for encryption and decryption operations (secret sharing). The asymmetric signing schemes supported by TCG are:

- RSASSA (S) and RSAPSS (S) depending from RSA;

- ECDSA (S), ECDAA (S), SM2 (A), ECSchnorr (S) and EDDSA (A) depending from ECC.

TPM 2.0 uses *symmetric encryption* for ciphering command parameters and objects stored in protected locations (memory outside the TPM). In the latter case, the encrypted objects have an HMAC calculated with the symmetric key, used for integrity checking. The HMAC is checked on encrypted data, in order to make performing power analysis more difficult [9]. The symmetric block ciphers supported by TCG are AES (S), TDES (A), XOR obfuscation (S, used only for confidential parameters), SM4 (A), CAMELLIA (A). Cipher FeedBack (CFB) mode is the only block cipher mode required by the specification, while the modes CTR, OFB, CBC, ECB are "A" classified. If a symmetric key is paired with an asymmetric one (as in ECC decrypting key), it is required that the two keys have the same number of bits as security strength. TPM 2.0 may implement also symmetric signature forms, that provide integrity protection over some data and the assurance that it comes from an entity that knows the value of a symmetric key. For symmetric signatures, TCG supports Cypher-based Message Authentication Code (CMAC) (A) using a symmetric block cipher algorithm, or the Hash Message Authentication Code (HMAC) (S) algorithm.

TPM 2.0 provides also *Hash functions*, used either directly by external software or in the processing of other TPM operations, as PCR *extend*. Hash functions are useful for integrity checking and authentication or as one-way functions (such as KDF). TCG supports the following hash functions: SHA1 (S), SHA256 (S), SHA384 (A), SHA512 (A), SM3_256 (A), SHA3_256 (A), SHA3_384 (A), SHA3_512 (A). The implemented hash function should have the same security strength as the strongest implemented asymmetric algorithm.

TPM 2.0 implements a *Random Number Generator (RNG)* module as source of randomness, for internal and external requests. It's a TPM's Protected Capability, whose random values are used for generating nonces, keys and randomness in signatures. Random numbers are produced as follows:

1. an internal *entropy source* (possibly more than one) generates entropy collected by a process, the *entropy collector*, that removes the bias;
2. a *state register* is updated with the collected unbiased entropy;
3. a *mixing function*, usually an approved hash function, uses the value in the state register to compute the random number.

The specification requires RNG to provide at least 32 bytes of randomness, more if the implemented internal functions require so.

With regard to *Key generation*, the TPM 2.0 can produce keys of two different types:

1. *Primary Keys*: they are computed starting from a given seed, generated by the RNG Protected Capability and persistently stored in the TPM for subsequent key generations (like the endorsement seed);
2. *Ordinary Keys*: they are computed starting from a seed generated by the RNG and different for each computation; the generated key is stored in a shielded location.

Key Derivation Functions supported by TCG are: KDF1_SP800_108 (S), KDF1_SP800_56A (S), KDF2 (A) [9] [10].

Authorization Subsystem

When a TPM 2.0 command is executed, if it accesses to shielded locations that need authorizations, the Authorization Subsystem is invoked by the Command Dispatch module:

- before the execution of the command in order to check, for each authorization, if it is of the right type and is valid for that object;

- after the execution of the command, to generate an acknowledge session value for the response.

The Authorization Subsystem requires the implementation of an hash function, the HMAC algorithm and an asymmetric algorithm (the latter is required only if TPM2_PolicySigned() is implemented) [9].

Random Access Memory

The RAM module holds TPM transient state, that is data that may be lost when TPM power is turned off. The values contained in the TPM RAM are in shielded locations, except those contained in the I/O buffer (if this is contained in a portion of the RAM). This memory contains data related to temporary state, authorization sessions and entities (keys and data objects loaded in the TPM from external memory) required for completing an implemented command. PCR banks are part of the TPM RAM [9].

Non-Volatile Memory

The NV memory module stores TPM persistent state. All the NV memory values are in shielded locations. NV memory can contain:

- structured data defined by the TPM 2.0 specification, that includes TPM's private data (hierarchy authorization values, seeds, keys, proofs) and data that can be read by a caller (counters, a clock and so on);
- unstructured data defined by a user or a platform specification.

The platform can access an NV memory location through an index (a handle) assigned to it, for this reason the location is called *NV index*. TPM 2.0 defines four types of NV indexes: *NV Ordinary Index* (already present in TPM 1.2), *NV Counter Index*, *NV Bit field Index* and *NV Extend Index* [7].

Power Detection Module

The Power Detection Module administrates TPM power states, which are ON and OFF power states. Since TPM power states should be directly related to platform power states, the TPM should be notified of all platform power state changes, so that [9]:

- any platform power transition that requires an RTM reset causes a TPM reset;
- any platform power transition that requires a TPM reset causes an RTM reset.

2.3.2 TPM 2.0 vs TPM 1.2

When the TCG worked on the TPM 2.0 specification, it considered the same design goals as for the TPM 1.2, adding to them several more. Moreover, changes in the TPM 2.0 architecture were needed to overcome some limitations arised with the TPM 1.2 [7].

Algorithm Agility

TPM 1.2 specification allowed only specific algorithms to be implemented in TPMs, SHA-1 as hash algorithm and RSA as asymmetric algorithm. TPM 2.0 specification overcomes this limitation and allows more flexibility in the type of algorithms that can be used. It does not enforce specific algorithms but refers to the TCG Algorithm Registry [10], that contains the list of algorithms to

which the TCG has assigned an *algorithm identifier* and that can be updated independetly by the TPM 2.0 specification. Algorithm agility [7] allows to implemet in the TPM a set of algorithms compatible with specific use cases, for example those compatible with legacy applications, with governments' requirements and so on. Moreover, if an algorithm becomes deprecated in the future because weakened by cryptanalysis, there will be no need for a new specification because algorithms can be easily upgraded to stronger ones.

Enhanced Authorization (EA)

The TPM 1.2 specification was very complex for what concerns the authentication mechanism for accessing TPM objects. TPM keys had two kinds of authorizations, one for using the key and another for migrating the key. Moreover, keys could be locked to specific localities, that is the software that originated a particular command, and to particular PCR values. NVRAM indexes also had two authorizations, one for reading and another for writing the index, and could be locked to particular localities and PCR values.

The TPM 2.0 specification extends and at the same time simplifies the previous authorization mechanism, unifying the way in which TPM objects can be authorized. The new policy authorization scheme constitutes the so-called *Enhanced Authorization* [7]. It is possible to acquire authorizations through proofs of identity, such as:

- *HMAC key*, useful for authorization in not trusted environments;
- *passowrd* in the clear, useful for the BIOS, where the added security of an HMAC key is not necessary;
- *signature*, for example through a smart card, possibly with additional information provided by a fingerprint reader, a biometric reader or a GPS.

The authorization can also be bound to the match of particular conditions, such as:

- *PCRs values*, to authorize access only when the system is in a trusted state;
- *localities*, for performing some operations only through a given secure software;
- certain periods of *time*, useful for allowing particular operations only during business hours;
- a range of values in an *NVRAM counter*, useful for limiting the usage of resources to a maximum number of times;
- *value of a bit in a NVRAM index*, useful to revoke access to a key for a given user;
- the fact that a given *NVRAM index* has been initialized;
- the *physical presence* of the user at the console of the machine, that requires BIOS-level confirmation for operations such as activating, deactivating, clearing or changing ownership of TPM.

All these authorization types can be combined through logical AND or OR operators for creating arbitrarly complex policies.

Quick Key Loading

In TPM 1.2 specification, the process for loading a key in the TPM was time-consuming since only asymmetric encryption could be used for protecting user's data. This limitation in the specification was due to the US export-control laws about cryptographic devices, to avoid the prohibition to export TPM implementations outside the US. So, when a user had to use a private key protected by the TPM, this key had to be decrypted with RSA algorithm by using another private key, the "parent" key, used to cipher user's key. To avoid having to perform this time-consuming process several times during a session, TCG adopted the solution to store in a "cache"

file an already loaded key, ciphered with a symmetric algorithm. This procedure speeded up the subsequent uses of the key since symmetric encryption was much faster than the asymmetric one. However, the problem about long delays for loading a key was not completely solved because, once the TPM was turned off, the symmetric key used to protect the cache file was erased and the next loading of the user's key required again a slow asymmetric operation.

The weakening of the export-control laws allowed the TCG to introduce in the TPM 2.0 specification the use of symmetric encryption for directly protecting user data, not only as an internal TPM's optimization. So, the TPM 2.0 implementations have a key loading time as long as that for recovering it from a cache file, and this quicker loading makes possible the use of the TPM by multiple users without they experiencing long delays [7].

Non-Brittle PCRs

The TPM 1.2 specification allowed an operation called *sealing*, that consists in authorizing the access to a key or a data only if a set of PCRs contain particular values. For example, we can seal a signing key to certain values contained in the PCRs related to the boot sequence, in order to enforce the usage of the key only if the system is in a trusted state. This operation, however, has the disadvantage to make tricky upgrading the system software because the measures calculated on it and stored in the PCRs change after the upgrading. For example, if we locked a key to a particular PCR 0 value, we had to unseal the key before upgrading the BIOS, then upgrade the BIOS and reseal it to the new PCR 0 value; if we didn't do so, the key had to be changed. This problem is known as *PCR fragility*.

With TPM 2.0 specification this problem has been solved giving the possibility to seal resources to PCR values signed by a particular authority, for example the OEM of the system software, instead of to a certain PCR value (although this is still possible). With this new capability we can lock data to be used only on systems that have any of the BIOS signed by the OEM, or any of the kernels signed by the OEM [7].

Flexible Management

In the TPM 1.2 specification only two kinds of authorization existed, the owner authorization and the *Storage Root Key* (SRK) authorization. Since the SRK authorization was typically 20 bytes of zeros, the owner role was used for many aspects that conceptually could be managed by different roles, like the privacy administrator or the platform manufacturer. The specification foresaw the possibility to delegate the owner role to different entities, but the command was difficult to use and required considerable NVRAM space, so almost no applications used this possibility.

The TPM 2.0 specification overcomes this problem by separating the different roles that were previously enabled by the owner authorization. This separation is realized by creating different *hierarchies* of objects, each one with its own authorization password and policy [11]:

- *Platform Hierarchy*: it's used by the platform manufacturer for controlling the integrity of the system firmware;
- *Storage Hierarchy*: already present in TPM 1.2 specification, it's used by the platform owner for controlling the access to the RTS;
- *Endorsement Hierarchy*: it's used by the privacy administrator for controlling the access to the EK, which represents the identity of the TPM;
- *Null Hierarchy*: it's used by any entity that want to employ the TPM as a simple criptographic coprocessor; differently from the previous ones, it doesn't require any authorization password and policy.

Finally, TPM 2.0 forsees specific authorization passwords and policies for resetting dictionary-attack counters [7].

Implementation Type	Security Level	Security Features	Relative Cost	Typical Application
Discrete TPM	Highest	Tamper resistant Hardware	$$$	Critical systems
Integrated TPM	Higher	Hardware	$$	Gateways
Firmware TPM	High	TEE	$	Entertainment Systems
Virtual TPM	High	Hypervisor	¢	Cloud Environment
Software TPM	N/A	N/A	¢¢	Testing & Prototyping

Figure 2.4. Different kinds of TPM implementations (source: [13]).

Identifying Resources by Name

In the TPM 1.2 specification TPM resources were identified by means of indirect references, the handles. Sigrid Gürgens found that this identification method exposed to a kind of attack. If two resources shared the same authorization policy, a user used the authorization believing to perform a given action, while it was possible to trick the low-level software by changing the handle to the resource, making it perform an action other than what the user wanted to perform.

The TPM 2.0 specification eliminated this attack identifying TPM resources by names cryptographically bound to them. Moreover, the name can be signed with a TPM key, thus providing a proof of its correctness [7].

2.3.3 TPM 2.0 implementations

The TCG called *Trusted Platform Module Library* the TPM 2.0 specification, in order to highlight the fact that the specification describes all the commands and the features that could be useful in a platform, leaving to the designers and developers to choose with more granularity the appropriate features and the required level of security for the targeted use case. This makes the TPM 2.0 specification much more flexible than the previous one, allowing TPM 2.0 to be used in a wide range of applications, from servers to PCs to embedded applications, including automotive, industrial and Internet of Things (IoT) contexts [12].

The most popular TPM 2.0 implementations nowadays are the following, summarized in figure 2.4:

1. ***Discrete TPM*** is implemented as single-chip component attached to the system using a low-performance interface (such as, Low Pin Count, or LPC). In this implementation, the TPM component has its own processor, RAM, ROM, and Flash memory; the only interaction with the system is via the LPC bus. The Discrete TPM provides the highest level of security, so it has to be designed, built and evaluated for resisting to tampering attacks. It is recommended for most critical systems.

2. ***Integrated TPM*** is still an hardware TPM but it is integrated into a chip that provides components implementing functions other than security. This kind of implementation has to be resistant to software attacks but it is not designed for resisting to tampering attacks. So, its security level is the next level down with respect to a Discrete TPM. Intel provides implementations of Integrated TPMs in some of its chipsets. Typically it is used for gateways.

3. ***Firmware TPM*** (fTPM), or *TEE TPM*, is implemented as protected software, that is as code running on the main CPU while it is in a special execution mode. The code executes in a protected execution environment, called a *Trusted Execution Environment (TEE)*, separated by the other programs that runs on the host system. In this case the TPM memory, that contains secrets like private keys, is a part of the system memory, partitioned by hardware, and it is only accessible when the host processor runs in this special mode. So, this implementation does not require a separate hardware. The host system can cause a change to the internal state of the TPM only through welldefined interfaces. The Firmware TPM provides a security level lower than the previous implementations because, in addition to lack of tamper resistance, its security level dependes on the TEE Operating System, bugs in the code that runs in the TEE and so on. Firmware TPMs have been implemented from Intel, AMD and Qualcomm.

4. ***Virtual TPM*** (vTPM), or *Hypervisor TPM*, is a software implementation of the TPM provided by an hypervisor to make TPM's capabilities available to Virtual Machines (VMs). It runs in an isolated execution environment with respect to the VM, in order to secure the vTPM code from that of the VM. In the hypervisor environment, each VM refers to its own instance of vTPM. Virtual TPMs provide a security level comparable to that of Firmware TPMs.

5. ***Software TPM*** is a TPM simulator that runs as a regular program within an operating system. It doesn't offer any security guarantee, since it is exposed not only to tampering but also to its own software bugs and to attacks addressed to a normal operating system. Nevertheless, it is very good for testing purposes, during the development of applications that interact with a TPM. There are many implementations of Software TPM 2.0, like IBM's "Software TPM 2.0" [14], based on the Microsoft's implementation of Part 3 and 4 of the specification. The project "TPM 2.0 Simulator Extraction Script" [15] instead contains a script that extracts the source code for the TPM 2.0 simulator directly from the PDF versions 01.16 and 01.38 of the Trusted Platform Module Library Specification.

2.4 TPM Software Stack 2.0

The ***TPM Software Stack (TSS) 2.0*** is a TCG standard specification designed to provide an high level API to programmers that develope applications accessing the TPM 2.0. The TSS is constituted by multiple software layers, represented in figure 2.5, that allow TSS implementations to scale from resource constrained embedded systems to high-level systems [16].

The TSS lower layer is the ***TPM Device Driver***, that is the driver of an operating system. It manages the communications with the TPM by reading and writing data in the TPM I/O buffer.

The ***Resource Manager*** handles the limited amount of TPM memory by performing a context swapping in a way similar to what the OS's virtual memory manager does for the system memory. It swaps the objects in and out of the TPM RAM so that multiple applications can perform TPM operations in parallel. This layer is transparent to the upper layers; if not implemented, upper layers have to implement its functionalities.

The next software level is ***TPM Access Broker*** (TAB), that has the purpose to manage multi-process synchronization for TPM applications. It guarantees an application that accesses TPM to be able to complete a TPM command without interference from other processes competing with it.

The ***TPM Command Transmission Interface*** (TCTI) interacts with the layers below it by managing two different interfaces to communicate with TPMs: the legacy TPM Interface Specification (TIS) and the command/response buffer (CRB). So, it can support all the different TPM implementations: local hardware TPMs, firmware TPMs, simulator TPMs, virtual TPMs and remote TPMs.

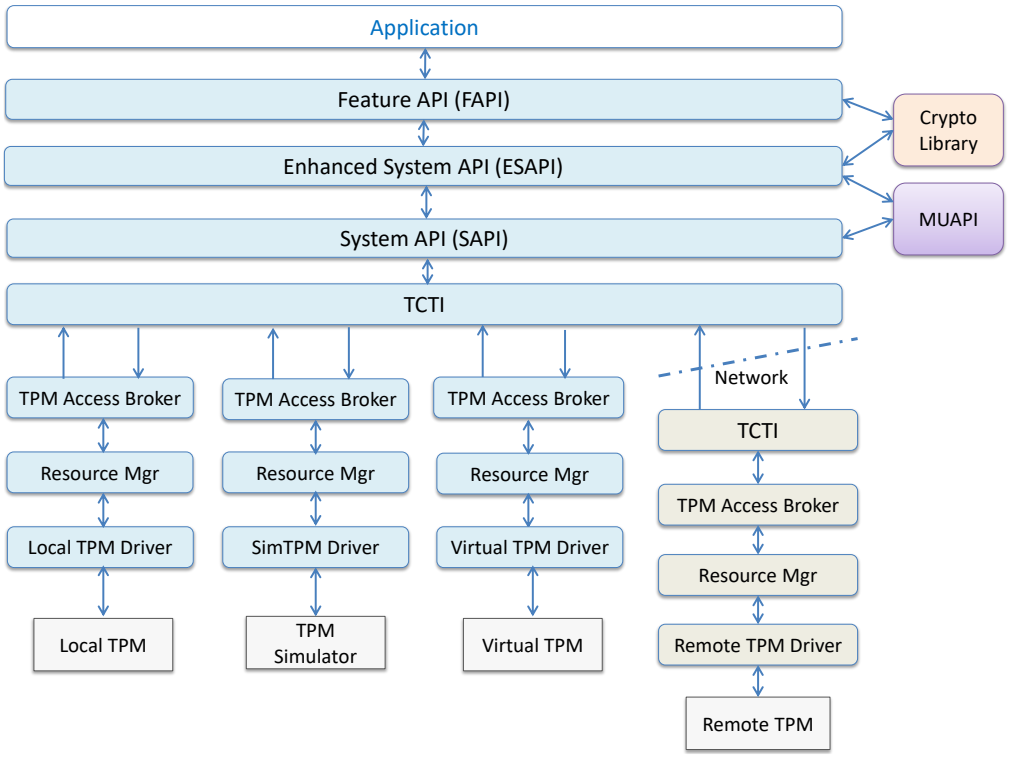

Figure 2.5. TSS 2.0 overview (source: [16]).

The ***System API*** (SAPI) provides access to all TPM 2.0 functionalities. It is a low level API, designed for "expert" applications that has to access the TPM, such as firmware, BIOS, OS, etc.

The ***Enhanced System API*** (ESAPI) is an interface designed to be directly above the System API. It has three primary purposes [17]:

1. providing an enhanced session management functionality on top of the basic SAPI functionality;

2. providing cryptographic functionalities for applications that want to encrypt data stream to the TPM, in order to protect against the probing of the data bus to the TPM;

3. providing 100% of the TPM functionality.

So, the ESAPI has the advantage to simplify some functionalities (managing sessions, calculating HMACs, encrypting and decrypting data) rather than the SAPI, allowing to reduce the programming complexity of an application accessing the TPM. However, differently from the ESAPI, the SAPI can be used in heapless environments, uses less RAM and its implementations have a smaller footprint (since it doesn't include cryptographic functions).

The ***Marshaling/Unmarshaling API*** (MUAPI) is used by the SAPI and ESAPI to perform the marshaling of the TPM command byte streams and the unmarshaling of the TPM response byte streams.

The ***Feature API*** (FAPI) provides higher level API to application programmers, allowing to write applications that uses the TPM without the need of knowing all low level details. It provides about 80% of the functionality of the TPM, differently from ESAPI that provides 100% of the functionality of the TPM.

2.5 TPM use cases

A TPM can perform the same cryptographic operations as an Hardware Security Module (HSM) or a smart card, so it can be used to solve all the problems that traditional cryptographic tokens solve. But, since it is directly attached to the motherboard of a platform, the TPM enables use cases that traditional HSMs couldn't perform at a low cost, like measuring the boot sequence of a platform for performing ***Remote Attestation (RA)***. Removable security tokens cannot be relied upon to be present before the OS is loaded, so they cannot be used for reporting on the boot sequence of the platform. In addition to Remote Attestation, software measurements stored in the PCRs can be used for sealing keys or other sensitive data stored in the TPM, authorizing their use only on a system whose software has not been tampered in any way. This allows to protect keys also from rootkits and bootkits.

Secure storage of private keys and cryptographic parameters can be used for trusted execution of cryptographic protocols. For example a Virtual Private Network (VPN) application can securely store an authentication key in the TPM and then perform authentication by interacting with the TPM device, without direct access to the private key.

Moreover, since a TPM is soldered to the platform, the strong identity established for the TPM with the EK certification is transferred in a transitive way also to the platform, and this can be used to ensure that a system is an authorized part of a cluster, while portable HSMs don't provide this property.

These TPM powerful features can be combined for providing more complex and powerful security functionalities to PC clients, servers and management systems [18].

2.6 Attacks against TPMs

The TPM has been designed by the TCG to provide a hardware-based Root of Trust to platforms, protecting cryptographic keys and sensitive data from software adversaries like malware, rootkits and physical adversaries. Most laptop, desktop computers, smartphones and embedded devices have a dedicated TPM chip or use a firmware TPM, since nowedays it is considered by computer scientists and IT industry at the base of Trusted Computing. However, over the years reserachers discovered several attacks that could be performed against some TPM implementations.

In 2010, Christopher Tarnovsky presented an attack against TPMs at the "Black Hat Briefings" security conference. This attack, performed for the Infineon SLE 66 CL PC, requires physical access to the chip, since it is necessary to remove the chip's case and top layer for inserting a probe to the internal bus. He was able to extract secrets from the TPM by spying on its communication after six months of work [19].

In 2015, among the documents that Snowden disclosed to the public there was one reporting that in 2010 a group of US CIA researchers declared at a secret annual conference called "Trusted Computing Base Jamboree" that they carried out a differential power analysis attack against TPMs, being able to extract TPM's secrets [20].

In 2017, a vulnerability, known as ROCA, was discovered in the code of a library developed by Infineon for its TPMs. The library generated weak RSA key pairs that let to infer the private key starting from the public conterpart. Consequently, cryptosystems that store these keys directly in the TPM without blinding allow an attacker to decrypt private data and impersonate their identity (identity theft and spoofing attacks) [21]. Infineon released firmware patches for its TPMs on October 10th 2017 [22].

In 2018, a group of reserchers of the National Security Research Institute described, at the "27h Usenix Security Symposium", two attacks against the TPM regarding the Power Detection module. These attacks allow an adversary to reset and forge the PCRs that store the measurements regarding the boot of the platform, by abusing of power interrupts that reset PCRs contents without causing the reset of the platform's RTM. The first attack exploited a design flaw in the TPM 2.0 specification regarding the Static Root of Trust for Measurement (SRTM). The second

attack exploited an implementation flaw regarding the Dynamic Root of Trust for Measurement (DRTM) in the "Trusted Boot" (tboot), an open source module used with Intel Trusted Execution Technology (TXT) to perform a measured boot of an Operating System. The countermeasure to these attacks needs hardware specific firmware patches [23].

Physical access to computers allows also *cold boot attacks*, a kind of side channel attack that allows an attacker to dump the contents of pre-boot physical memory to retrieve the private key used for disk encryption [24].

In 2020, D. Moghimi, B. Sunar, T. Eisenbarth and N. Heninger published a paper, presented at the "29h Usenix Security Symposium", where they describe the timing leakages discovered on Intel fTPM (FIPS 140-2 certified) as well as in STMicroelectronics' TPM chip (Common Criteria EAL4+ and FIPS 140-2 level 2 certified). Their analysis reveals that both devices exhibit secret-dependent execution times during cryptographic signature generation based on elliptic curves, allowing an attacker to apply lattice techniques to recover 256-bit private keys for ECDSA and ECSchnorr signature schemes. On Intel fTPM, the private key is recovered after about 1300 observations and in less than two minutes, while on STMicroelectronics' TPM chip the private ECDSA key is extracted after fewer than 40000 observations.They performed also a remote attack against StrongSwan IPsec VPN, that uses a TPM to perform digital signatures for authentication, recovering the server's private authentication key by observing 45000 authentication handshakes on the network connection. The reserchers discovered these attacks at the biginning of 2019 and informed both Intel and STMicroelectronics about their findings. Intel issued patches to correct this issue on November 12th 2019, while STMicroelectronics updated their TPM chip making it resistent to this kind of attacks on September 12th 2019 [25].

Chapter 3

Integrity Measurement Architecture (IMA)

As stated in chapter 2, TCG's specifications [8][9] expose all the concepts that are at the base of Trusted Computing philosophy, defining the RoTs that a Trusted Platform has to implement. TCG's specifications define the building blocks on which a Trusted Boot is based, in particular the fact that the CRTM and the *transitive trust* concept allow to establish an RTM that measures the system boot components and stores the measures in the RTS (the PCRs in the TPM), permitting to an external entity to verify that the system booted in a secure way. However, TCG's specifications are operating system agnostic, they do not state how the RTM of the platform extends in the OS. *Integrity Measurement Architecture* (IMA) is the Linux kernel's implementation of the integrity measurement system conceinved by the TCG and it allows to extend the chain of trust from the BIOS up to the application layer, as shown in figure 3.1. IMA extends the principles of Trusted Boot and Secure Boot to the Linux kernel, thus resulting in an essential part of the TCB of a Trusted Platform. It is part of the *Linux Integrity Subsystem* since 2009 starting from version 2.6.30 and is currently one of the most accepted TCG-compliant solutions for measuring dynamic executable contents [11]. IMA measures all the executables, configuration files and kernel modules as soon as they are loaded onto the Linux system before passing the control to them, and extends these measures in the TPM. This permits an external entity to verify not only the boot of the system, but also which applications and kernel modules have been loaded in the platform, if they are expected or undesirable invocations, if the software has a trusted state and if its configuration is as expected. All this can be done without requiring a new CPU mode of execution or a new operating system, but merely relying on the hardware RoT provided by the TPM, which is nowadays ubiquitous in all platforms [26].

While in the boot process the modules to be measured and the sequence in which they are executed is predetermined, in an operating system there is large variety of software components to be managed (e.g., binaries, shared libraries, scripts, kernel modules) and the order in which they are loaded is not predictable. For this reason, in order to let an external entity to attest the state of the platform during runtime, the measures collected by IMA cannot be only stored in a PCR otherwise the external entity could not be able to check if the measurement aggregate, contained in the PCR, represents or not a trusted state. Instead, a Measurement Log (ML) file is used in order to store the sequence of the Measurement Events (MEs) as they occurred in the system and a PCR in the TPM (typically PCR 10) is used to protect the integrity of this ML. In order to attest the system state at runtime, the external entity can use the content of the IMA PCR for checking that the ML has not been tampered with; if the ML is valid, the remote entity analyzes it entry by entry, in order to determine if the change in the system state represented by each ME is trustworthy.

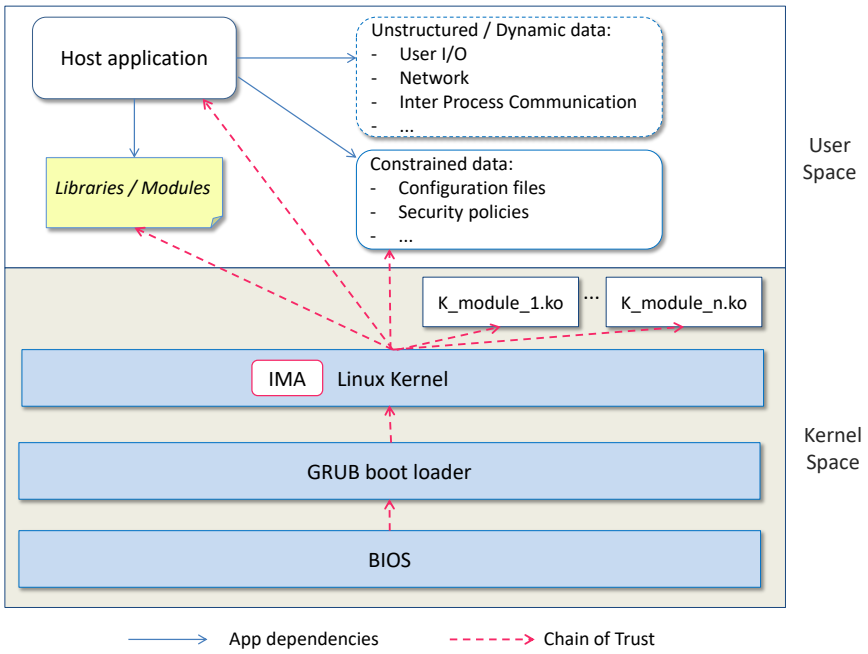

Figure 3.1. Chain of trust from BIOS to the application layer (source: [26])

3.1 Measuring System Integrity

The main goal of IMA designers was to let a remote entity (*challenger*) to verify that an application running on another system (*attesting system*) has a sufficient integrity level to be used. The *integrity* of an application is a binary property that indicates if the application and/or its execution environment have been modified in an unauthorized manner, which may cause the application to have an incorrect or malicious behaviour so that the challenger cannot rely on it [26]. In order to establish if an application or its environment has been tampered with, it is fundamental to understand what are the elements that can affect the integrity of the application or its environment and how they can be monitored. The platform starts by booting the operating system and, in order to have a trusted OS, the system has to support the *Trusted Boot* process 2.2.4, so that a challenger is able to determine whether all the boot components meet the desired integrity requirements. Once the kernel is booted, user-level programs can be executed. Since all the executable content related to a program impacts its integrity level, it is fundamental to measure all information of this type, such as the main executable file, all the libraries used by the executable and, if the application is an interpreter, such a bash, all the script files that are read as normal files but then are executed. Besides the files with executable content, there are other files that affect the execution behaviour of the application and should be measured to establish the application's integrity level, such as configuration files, security policy files, in general all files that contain constrained "structured" data whose measure can be compared to reference values. The integrity of an application also depends on dynamic "unstructured" data that is consumed by running executables, such as data coming from remote clients, other applications, files and so on. Differently from "structured" data, "unstructured" data measurement is useless because it is difficult for a challenger to predict all possible values that do not subvert the application's integrity. So, data affecting the application integrity can be devided in two categories:

- *high integrity code and data* is all the information (executable code and "structured" data) whose measure can be used to state the application's integrity level;
- *low integrity data* represents dynamic "unstructured" data whose measure cannot be used by a challenger for determine the application's integrity level.

The IMA measurement module was designed taking into account the principles exposed by

Clark and Wilson at the "IEEE Symposium on Security and Privacy" in 1987 [27]. An integrity verification procedure of an application [26] should first determine the *verification scope*: if the information flows in which the application is involved comply with a "mandatory policy", then it is sufficient to check the state of integrity of the application itself and of all the processes on which it depends; otherwise, the integrity level of the application involves measuring all processes running on the system. What needs to be measured of a process is: all its *executable content*, including modules loaded by the kernel, libraries loaded by the dynamic loader and code loaded by the process itself; all *structured data*, that is data having defined integrity semantics. Instead, *unstructured data* does not have identifiable integrity semantics, so it can not be measured; its integrity can be considered dependent by the integrity of the processes that managed it, or it can be determined by data history and security policies; another possibility is to make unstructured data undergo *transformation procedures* that upgrade low integrity data to high integrity data. Moreover, for the challenger to correctly verify the application's integrity level in a given moment, the measurement list has to be:

- *fresh*, that is not subject to replay attacks;
- *complete*, that is it has to include all the measurements performed up to the time when the attestation is executed, so the truncation of the list for hiding a corrupted state has to be detected;
- *unchanged*, the measurements contained in the list regarding executable code and structured data have not been tampered with and any modification has to be detected.

The previous analysis shows that measuring the integrity level of traditional flexible systems is a complex problem that has to be divided into several coordinated tasks. IMA designers addressed this problem with the aim to provide a tool able to identify integrity bugs and form a basis for constructing reasonable system's integrity verifications [26].

3.2 IMA Design

The goals of the Linux kernel integrity subsystem are [28]:

- *detecting* accidental or malicious file changes, both remotely and locally;
- *appraising* a file's measurement against a trusted value;
- *enforcing* local file integrity.

These goals require implementing in the kernel the functionalities *collect*, *store*, *attest*, *appraise*, *audit* and *protect* of file measurements. These goals are achieved in the Linux kernel by means of the IMA module, which consists of the following major components:

- ***IMA Measurement*** is responsible, on the attesting system, for determining what files to measure, performing measurements on files and maintaining them in a secure way; by means of the *Integrity Challenge Protocol* and the *Integrity Validation Mechanism*, it enables a challenging system to perform the *Remote Attestation* of the entire software stack of the attesting system; in particular, the *Integrity Challenge Protocol* enables authorized challengers to retrieve the measurement list and to validate its freshness and integrity; while the *Integrity Validation Mechanism*, performed by the challenger, verifies that all the measures contained in the measurement list are representative of code and structured data in a trusted state;
- ***IMA Appraisal*** is responsible for locally comparing file measurements against trusted digests, stored in the file's security extended attributes, possibly denying the access to the file in case of measurement mismatch;

- ***IMA Audit*** is responsible for including the file's name and measurement in the system audit logs, that can be used for system security analytics/forensics.

Listing 3.1. IMA components invoked in the process_measurement() function

```
static int process_measurement(...) {
    ...
    rc = ima_collect_measurement(...);
    ...
    if (action & IMA_MEASURE)
        ima_store_measurement(...);
    ...
    if (rc == 0 && (action & IMA_APPRAISE_SUBMASK)) {
        ...
        rc = ima_appraise_measurement(...);
        ...
    }
    if (action & IMA_AUDIT)
        ima_audit_measurement(...);
    ...
}
```

IMA Measurement, IMA Appraisal and IMA Audit complement each other, but can be configured and used independently of each other [28]. Listing 3.1 shows that the `process_measurement()` function, defined in the kernel source file ima_main.c [29], invokes the IMA components independently of each other, on the basis of the `action` that has to be executed on the file and that depends on the configured *IMA Policy* (described in section A.1.4). The Linux Integrity subsystem comprises also the *Extended Verification Module (EVM)*, which is responsible of protecting the file's security extended attributes.

3.2.1 Assumptions

IMA design is based on some assumptions about the threat model and the platform configuration, without which it would be possible attackers to be able to trick a remote client. IMA uses services and protections specified in the TCG standards in order to provide to challenging parties the mechanisms to prove platform identity and integrity protection of the measurement list. For this reason, IMA should be used on platforms where a hardware TPM, compliant with TCG's specifications, is installed, ensuring that the measurements of the boot components and of the runtime system are properly taken and stored with a hardware-rooted chain of trust. The threat model comprises all kinds of software attacks but not direct hardware attacks against the system, since not all TPM implementations resist against physical attacks.

The measurements of the code and structured data are assumed to be representative of their identity, consequently of their integrity state. Self-changing code can also be evaluated through its measurement since the ability of the code to change itself is reflected in the measurement. This is true also for the kernel code, that changes itself through loading and unloading of kernel modules. Kernel changes performed by overwriting kernel code with malicious DMA transfers are not addressed, although the code that sets up the DMA is measured and so it can be evaluated.

The challenging parties are assumed to hold an AK certificate, emitted by an Attestation CA, binded to an AK_{pub} identity key generated by the attesting system's TPM, in order to prove that the quoted PCRs belong to the TPM of that attesting system, before they are used to evalute the system boot and the integrity of the measurement list.

No confidentiality requirement is assumed on the data of the measurement list, other than those that can be achieved by controlling access to the attesting system.

Finally, the challenger system is assumed to safely evalute the trustworthiness of the measurements received by the attesting system, either by comparing them to a list of trusted measurements (whitelist), or by using measurements signed by trusted parties according to common policies [26].

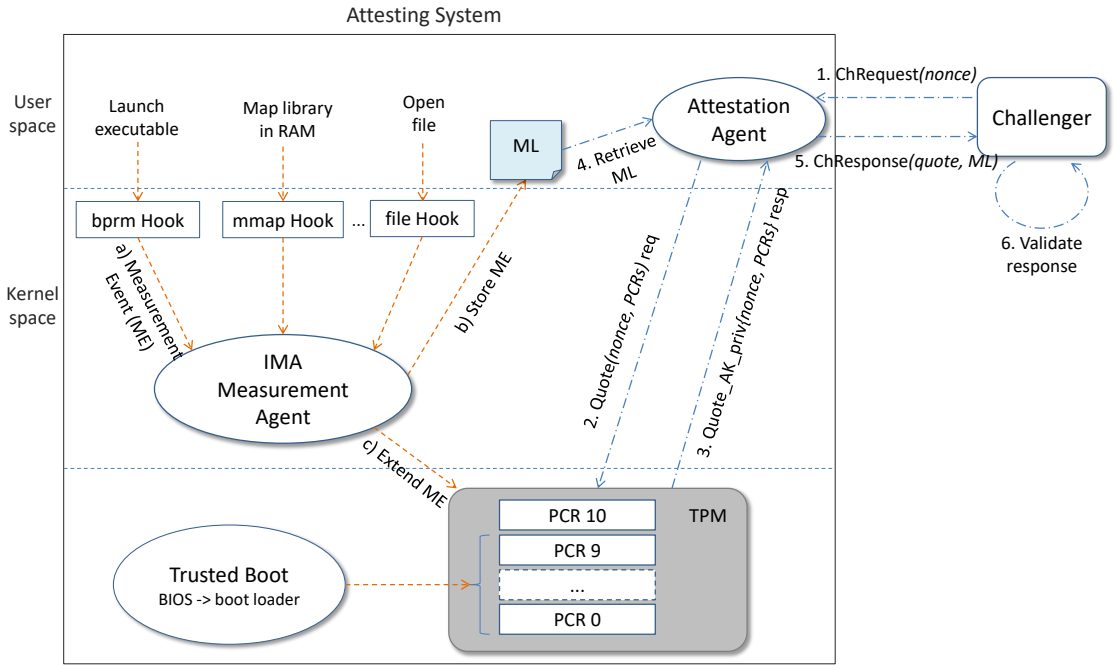

Figure 3.2. Integrity Measurement Architecture (source: [2])

3.2.2 Remote Attestation

Before entering into the details of the different IMA components, it is worth to describe how they interact to enable Remote Attestation (fig. 3.2). The IMA Measurement mechanism starts when an *IMA Hook* in the attesting system receives a ME (a), such as loading a binary program, mapping a file in RAM or opening a file for reading or writing. It is responsible for measuring the received ME, that is for calculating the hash value of the file's content with a proper secure hash function. Then IMA Measurement stores the file digest, together with other file metadata, in an ordered list of MEs in the kernel (b) and extends a digest computed over the ME into a PCR in the TPM (typically PCR 10) through the *extend* operation (c).

This measurement mechanism enables a remote challenger (also called *Verifier*) to validate the integrity state of the attesting system by performing the *Remote Attestation*. The integrity challenge starts when a challenger sends a challenge request to the *Attestation Agent*, specifying a *nonce* for guaranteeing freshness of the response (1). When receiving the request, the Attestation Agent requests to the TPM a quote, containing the *nonce* sent by the challenger and the PCR values (2) (typically PCRs related to the boot and PCR 10 containing the IMA measurements aggregate); the TPM sends back the quote, signed with the AK_{priv} identity key (3). Then, the Attestation Agent retrieves the IMA ML (4), creates an *Integrity Report* (IR) containing the quote and the IMA ML and sends it to the Challenger (5). The Challenger validates the received IR, verifying that the quote is fresh and authentic, the ML is non-tampered and the measurements contained in the ML represent a trust system (6).

3.2.3 IMA Measurement

IMA Measurement extends the principle of the *Measured Boot* into the operating system, so that BIOS measures the bootloader, the bootloader measures the initial kernel code and the kernel, enhanced by IMA, measures changes to itself (kernel module loads) and to the application layer. IMA Measurement handles a linked list of measures in the kernel memory. This list is called *measurement list* and contains all the MEs that represent the integrity history of the attesting system. The first node of the list is always the `boot_aggregate`, containing the digest computed

over the PCRs with indexes from 0 to 7. If the attesting system does not have a TPM chip, the measurement associated to the `boot_aggregate` is all-zeros. IMA calculates the `boot_aggregate` digest in this way:

1. it reads the PCRs from the bank with hash algorithm equals to that configured for measuring the files;
2. if the TPM does not have a PCR bank with the hash algorithm used for measuring the files, it reads PCRs from the bank required by the TCG, that is, SHA-256 for TPM 2.0 and SHA-1 for TPM 1.2;
3. if the SHA-256 bank is not found, it reads the PCRs from the SHA-1 bank also for TPM 2.0.

After the `boot_aggregate`, IMA Measurement processes all the accessed files that match one of the `measure` rules configured in the IMA policy and determines if that file needs a new measurement. In particular, IMA performs a new measurement over the file contents only if:

1. the file was never measured yet;
2. the file changed since last measurement;
3. the kernel has no way of detecting changes for that file.

For IMA to dectect file's changes, the filesystem needs to be mounted with the *i_version* option. Starting from Linux kernel version 4.17, *i_version* is considered an optimization so, if *i_version* is not enabled, either because the local filesystem does not support it or the filesystem was not mounted with *i_version* support, the file will always be re-measured, even when it did not change. Supposing that the file has to be measured, the measurement is performed by calculating a digest over the complete contents of the file. The digest is computed with a secure hash function which for default is SHA-1; the kernel command line parameter `ima_hash` allows to specify another hash algorithm among those supported by the Linux kernel, listed in the /crypto/hash_info.c kernel's source file.

After having measured the file, IMA determins if a new node has to be added to the measurement list, with the criterion that a new node is added only if it is not already contained in the measurement list. For doing this, IMA manages also a hash table containing the nodes inserted in the measurement list, so that the check on the nodes is faster. This behaviour, referred to as *IMA caching* mechanism, allows to keep the size of the measurement list as small as possible. In the case in which the new node has to be added to the measurement list, the hash computed on this node is extended into the IMA PCR in the TPM. This makes any modification to the measurement list visible to challenging parties during the Remote Attestation process. Moreover, the *extend* operation is performed before the measured component takes the control of the platform, directly as executable or indirectly as data file, so a potentially corrupted component does not have the possibility to extend a measurement of itself that does not correspond to its real state. Although a corrupted component, ones gained the control of the platform, may perform other extensions in the IMA PCR, the properties of the secure hash algorithms guarantee that it is not possible to adjust the aggregate in the IMA PCR so that it represents a trusted system. Thus, malicious components can tamper with the measurement list, but the tampering is detectable by recomputing the aggregate on the measurement list and comparing it with the aggregate stored in the IMA PCR [26].

Afterwards, each node of the measurement list is recorded in two ML files both located in the `securityfs`, one in ASCII format named `ascii_runtime_measurements`, the other in binary format named `binary_runtime_measurements`. The writing of these files does not occur contextually to the measurement process, it is performed by a different task. The information contained in the ML files depends on the selected *IMA template*.

PCR	template-hash	template-name	filedata-hash	filename-hint
10	b926a8e790[...]36ed834d6a	ima-ng	sha1:999cdeec[...]e5fc0417	boot_aggregate
10	10db6ece69[...]8d509963e0	ima-ng	sha1:918df370[...]a10a3334	/usr/bin/kmod
10	60ef4670f3[...]35bec9becd	ima-ng	sha1:12b731d8[...]e809904	/usr/lib/.../ld-2.31.so
10	513c0f28b2[...]116be9f7b6	ima-ng	sha1:d850f87b[...]c8ceb243	/usr/lib/.../liblzma.so.5.2.4
10	066b83cc87[...]c57190e4b1	ima-ng	sha1:3d1c175a[...]d7136ea	/usr/lib/.../libcrypto.so.1.1
10	6200cc7ab4[...]bbe74eab98	ima-ng	sha1:a1ca4cd2[...]1d84c1d6	/usr/lib/.../libc-2-31-so
...	...	...	...	...

Figure 3.3. Example of IMA ML created with `ima-ng` template.

IMA Template Management Mechanism

Each entry in a ML contains the information that represents a ME. An *IMA template* specifies what kind of information regarding the ME has to be recorded in the IMA measurement list and consequently shown in the ML files. The first IMA template, called `ima`, contained only two fields for each entry: the file digest limited to 20 bytes (computed with SHA-1 or MD5 algorithms) and the name of the file limited to 255 characters. To overcome this limitation and allow ML entries containing additional file metadata, the template management mechanism was introduced starting from Linux 3.13.0. The core of this mechanism consists of two data structures:

- a *template field*, that defines a type of data that can be stored in a ME;
- a *template descriptor*, that states all the template fields that a ME will contain.

The figure 3.3 shows an example of ML created with the default template `ima-ng`. From left to right, the fields represent:

1. the index of the *PCR* in which the entry was extended, in this case PCR 10;
2. the *template-hash*, that is the digest extended in the PCR with index specified in the first field; the digest is computed over the *template fields* with the SHA-1 algorithm;
3. the *template-name* used for the entry;
4. the *eventdata-hash*, that is the hash computed over the file's contents or the boot-PCRs contents in the case of `boot_aggregate`; in the figure the algorithm used is SHA-1, which is the default hash algorithm;
5. the *event-name*, that is typically the file pathname.

3.2.4 Integrity Challenge Protocol

The Integrity Challenge Protocol has a fundamental role in the Remote Attestation process and describes how a challenger can securely retrieve the attestation information from the attesting system. In this section some aspects of the protocol are examined in more detail, particularly how it protects the Integrity Report (IR) from the major threats.

A malicious attesting system could try a ***replay attack*** by sending back to the challenger an IR (containing the IMA ML and the TPM quote with the IMA PCR aggregate) created before the system was corrupted. The challenger protects itself against this kind of attack in this way: it makes an IR request tied to a non-predictable random *nonce* of 160 bit, then when it receives the response from the attesting system, it verifies the freshness of the IMA PCR aggregate by checking that the quote has been computed with the nonce that it sent in the request and, if the

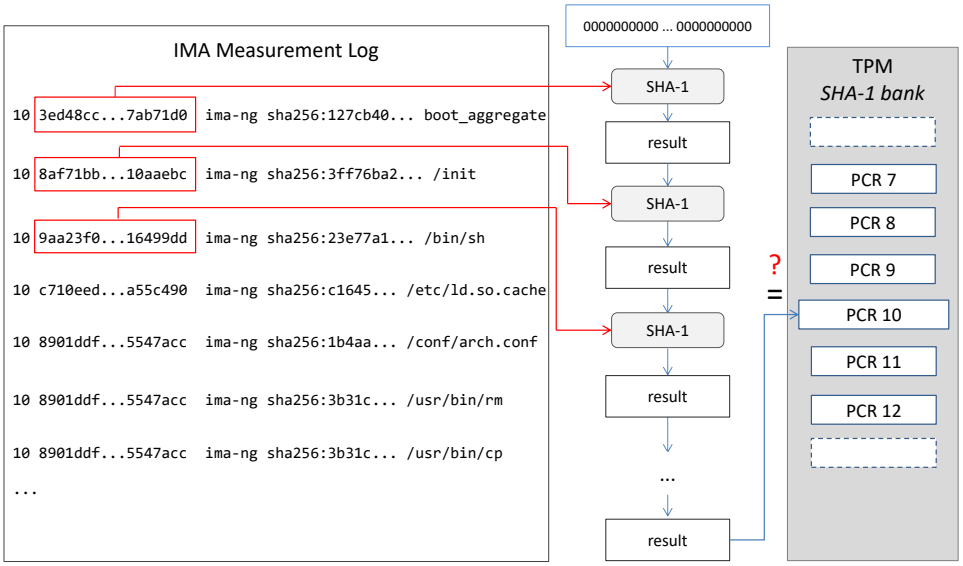

Figure 3.4. ML verification process (source: [11])

nonce does not match, it rejects the IR. Freshness of the quote is guaranteed as long the nonce is *unique* and *unpredictable*. If the attesting system receives the same nonce twice, it could respond to the challenger with old measurements withouth being discovered, for this it is necessary the nonce to be *unique*. Moreover, if the attesting system could predict a small enough set of values in which the nonce most likely will fall, it could collect TPM quotes using predicted nonces and afterwards respond to the challenger with them, providing a valid quote that does not reflect the current status of the system, for this it is necessary the nonce to be *unpredictable*. If the mechanism of nonce generation doesn't provide adequate security, then the validity of the AK key could be restricted in time, making collected quotes unusable for replay attacks.

A malicious attesting system, or an intermediate attacker, could try a ***tampering attack*** with the ML and the IMA PCR aggregate, before or during their transmission to the challenger. The challenger detects the tampering with the IMA PCR by validating the quote signature performed by the TPM of the attesting system: if the attesting system, or an intermediate attacker, changed the IMA PCR aggregate, this tampering will invalidate the signature. Moreover, the challenger can detect tampering with the ML by walking through the entries contained in the ML and recomputing the IMA aggregate as performed by the TPM. The figure 3.4 shows the process of verifying the integrity of the ML performed by the challenger. Supposing that i MEs have been stored in the ML, the IMA PCR aggregate can be recomputed by performing the *extend* operation with the template-hash th_i of each ML entry:

$$result = SHA1(...SHA1(SHA1(0 \,||\, th_1) \,||\, th_2)... \,||\, th_i)$$

If the recomputed aggregate matches the signed one, it means that the ML is authentic and can be used for evaluating the integrity state of the platform; otherwise it is invalid and has to be rejected.

A malicious attesting system, or an intermidiate attacker, coud try a ***masquerading attack***, by replacing the original attestation information with the ML and the IMA PCR aggregate generated by another non-compromised system. The challenger can discover a cheating system by verifying the quote signature with the AK_{pub} key binded to the attestation certificate, issued by an Attestation CA, that guarantees the identity of the attesting system. If the signature verification fails, the quote is not authentic and has to be rejected. Obviously, the attestation certificate must be verified to be valid by checking the Certificate Revocation List (CRL) issued by the Attestation CA.

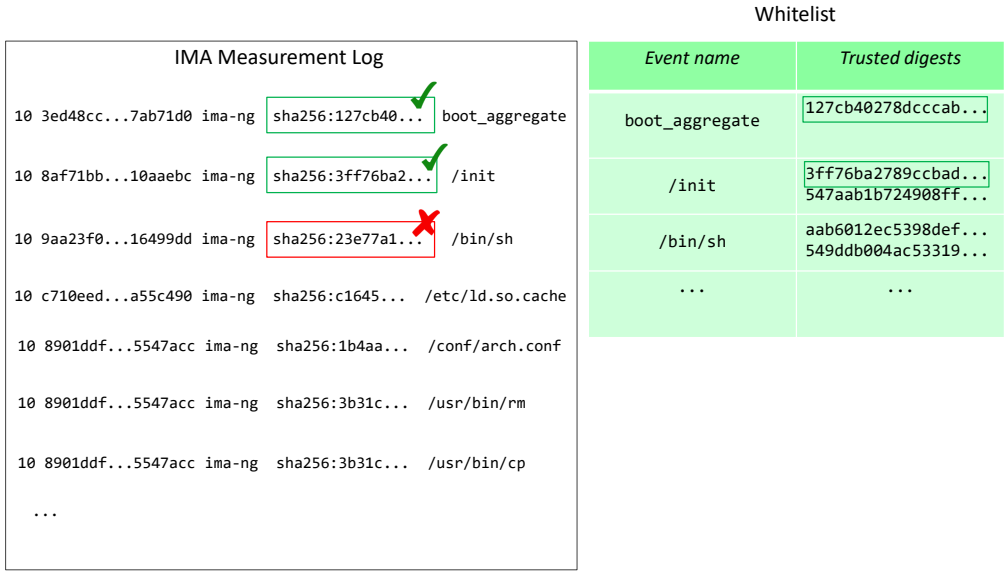

Figure 3.5. ML measurements validation

3.2.5 Integrity Validation Mechanism

In order to evaluate the trusted state of the attesting system, the challenger has to evaluate the file measurements contained in the ML entries, by comparing them with a list of trusted values (called *whitelist*). The validation of measurements related to executable files is logically the same as for measurements of data files. Entry by entry, the challenger takes the event name (i.e. the file path-name) and retrieves from its whitelist the set of valid measurements for that event. If the challenger doesn't find the file path-name in the whitelist, it means that an unknown program has been executed on the attesting system. Otherwise, the challenger compares the measurement contained in the ML entry with those contained in the whitelist, as shown in figure 3.5. If the measurement matches one contained in the whitelist, the file has a known integrity state which does not alter the trustworthiness of the platform. If the measurement does not match the whitelist contents, it means that the file could be an updated version of the program, or its code or data has been manipulated by an attacker. The challenger must have a policy that establishes what actions to take when unkwnon file names or untrusted measurements are detected in the ML. Usually, a distrusted measurement leads to evaluate untrusted the whole attesting system, unless additional isolation mechanisms guarantee the isolation of the detected untrusted executable.

The integrity of the attesting system can be monitored over the time by periodically repeating the Remote Attestation process: as long as the attesting system is evaluated as trusted every time the process is repeated, we could say that the system has never been tampered with. However, in order this to be true, the measurements evaluated in the various attestations should belong to the same *epoch*, that is, any system change occurred after an attestation should have been recorded and be visible in the following attestation. However, if the attesting system is compromised after an attestation and is rebooted before the following one, it results trusted after both attestations because the reboot hid its untrusted state to the challenger. So, in order to be sure that the system has always been trusted, it's important to implement a mechanism to discover if the epoch changes, that is, if the system rebooted between two attestations. A way to do this is by using TPM counters, a kind of NVRAM indexes that can be only increased throughout their lifetime, never reset or decreased. Each time the system reboots, the BIOS could increment a TPM counter so that, adding the counter value, signed by the TPM, to the attestation information, the challenger can detect if the attesting system rebooted between two consecutive attestations, because in this case the value of the counter would be different. While, if the counter remains the same, we can be sure that the ML offers a retrospective view of everything happened to the attesting system up to that point [26].

3.2.6 IMA Appraisal

IMA Appraisal is part of the Linux kernel starting from version 3.7 and extends the principle of the *Secure Boot* into the operating system. While IMA Measurement enables the Remote Attestation process, IMA Appraisal enables a local validation of the file integrity validation by comparing the file measurement against a trusted value stored in an extended attribute of the file. This attribute, named `security.ima`, can contain either a digest of the file for evaluating its integrity, or a signature of the file for evaluating its integrity and authenticity. IMA Appraisal is disabled by default so, in order to use it, it is necessary to activate it in the kernel configuration file and then to build and install the new kernel.

3.2.7 Extended Verification Module (EVM)

EVM is part of the Linux Integrity subsystem since version 3.2 of the kernel and is responsible for detecting offline tampering of the standard security extended attributes of the files. The possible attributes are:

- `security.ima`, which contains the file's trusted hash used by IMA Appraisal to take decisions;
- `security.selinux`, which contains the file's SELinux label;
- `security.SMACK64`, which contains the file's Smack label;
- `security.capability`, wihch contains the file's capability label.

In addition to these security extended attributes, EVM can be configured to protect other information:

- newly defined SMACK xattrs: `security.SMACK64EXEC`, `security.SMACK64TRANSMUTE` and `security.SMACK64MMAP`;
- the filesystem UUID;
- all the xattrs specified in the file `/sys/kernel/security/integrity/evm/evm_xattrs`.

EVM protects the set of attributes by using another extended attribute called `security.evm`, which can contain:

1. the HMAC computed on the xattrs, keyed with an `evm-key` loaded on root's keyring;
2. the digital signature of the xattrs.

For allowing EVM to check digital signatures, it is possible to load onto the `.evm` trusted keyring the RSA public key or the X509 certificate containing the public key to be used for verification. The X509 certificate path has to be specified in the kernel configuration file.

3.2.8 IMA Audit

IMA Audit, available in the Linux kernel since version 3.7, augments the kernel audit subsystem by adding IMA specific records used to assist with security analytics/forensics. It is enabled specifying rules with the `audit` action in the IMA policy file. For example, in order to audit all executed programs, the rule can be specified as follows:

```
audit func=BPRM_CHECK
```

The default policies of the Linux kernel do not include `audit` rules.

Chapter 4

Container Attestation

Virtualization is the virtual simulation of something that is real. For a long time, in computing this concept was tied to *hardware virtualization* or *platform virtualization*, which allows the abstraction of the computer hardware and for this reason we talk about *virtual computer* or *Virtual Machine (VM)* [30]. This means that every piece of hardware, including processors, memory and peripheral devices, is emulated by a software layer, called *hypervisor* or *Virtual Machine Monitor* (VMM), which enables the execution of an Operating System (OS), called *Guest OS*. A VM accesses the physical hardware, for example to communicate over the network, through interfaces provided by the hypervisor. The *virtualization* concept has a fundamental role in the cloud computing, since it enables to optimize the use of the overall hardware resources and to facilitate service management, improving flexibility, availability, and lowering costs.

However, in recent years a new concept of virtualization has emerged, when developers looked for more efficient alternatives to VMs for answering the question: why virtualize an entire machine when it would be possible to virtualize only a small part of it? Especially if an application, like the simple services that are typically deployed on the cloud, does not need of all the functionalities that a VM provides. Working on this problem, Google developed new kernel features (*namespaces*, *cgroups* and *capabilities*), that were included in the Linux kernel since 2008 and led to the development of a kind of virtualization, the *containerization*, which provides a higher level of abstraction than a VM, raising it from the hardware level to the operating system level. In fact, containers do not have their own virtualized hardware but, directly communicating with the host kernel, use the hardware of the host system. For this reason, this kind of virtualization is called *OS-level virtualization*. The advantage of a container compared to a VM lies in the lightness

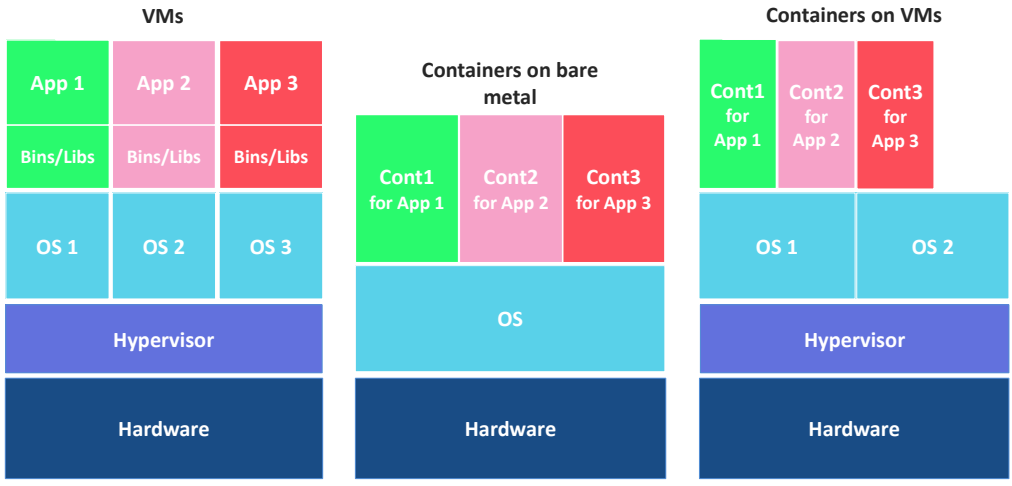

Figure 4.1. Virtual Machine and Container deployments (source: [31])

because, not having its own operating system, it has a weight of a few tens of MB, against the GB of a VM and this means lower demand for computing power and higher speed, not having the overhead of the hypervisor. This explains why the OS-level virtualization is also known as *lightweight virtualization* and the containers as *light VMs*. "Containerized" services can be run or added, removed or upgraded in a fraction of second, facilitating the application management and enhancing the customer experience. A study commissioned by Red Hat to TechValidate, involving more than 383 IT managers and operators from around the world, highlighted that 67% of respondents were considering the introduction of container-based applications in their business plans; at the same time, respondents had also expressed concerns on issues about security, certification and skills [32]. In recent years there has been a considerable growth of container-based applications in the cloud thanks to all the advantages this technology offers; for this reason, it is fundamental to address the study about the security problems related to containers usage and find solutions for keeping security risks low. In particular containers, providing less isolation than VMs, are more vulnerable to attacks and expose the host system to privilege escalation, so monitoring the integrity state of software and data related to containers and the underlying host platform is crucial. Remote Attestation based on TCG's standards works very well to attest physical platforms; it has been adapted to VMs attestation, even if many problems still need to be solved for efficiently providing a hardware-root of trust; while a way to apply Remote Attestation to containers is a challenge still opened to researchers, who are presenting various kinds of solutions to the scientific community for building container trust. In this chapter, the major solutions developed in the last years in this reserach field will be illustrated.

4.1 Container-based Virtualization

This section focuses on Linux's container-based virtualization, which is rooted in the kernel features *namespaces*, *cgroups* and *capabilities*.

A ***namespace*** abstracts a global system resource, making it appear to processes belonging to that namespace as they have their own instance of the global resource, isolated from processes not belonging to that namespace. Changes to the global resource are visible only to processes within the same namespace, while are invisible to other processes. Namespaces are the foundation of process isolation for implementing container-based virtualization. The following list shows the namespace types available in Linux since kernel version 2.6.26 [33]:

- a *cgroup namespace* isolates the root directory of control groups;
- a *IPC namespace* isolates System V Inter Process Communication (IPC) and POSIX message queues;
- a *network namespace* isolates network devices, stacks, ports, etc.;
- a *mount namespace* isolates mount points;
- a *PID namespace* isolates process IDs, so that it is possible to create processes in a PID namespace with a PID already in use in another PID namespace;
- a *time namespace* isolates boot and monotonic clocks;
- a *user namespace* isolates security-related identifiers and attributes, in particular user and group IDs;
- a *UTS namespace* isolates hostnames and Network Information Service (NIS) domain names.

Control groups, typically referred to as *cgroups*, allow to organize processes into hierarchical groups in order to limit and monitor the usage of several kinds of resources. A cgroup *hierarchy* is defined through a pseudo-filesystem called *cgroupfs*. Each level of the hierachy can have attributes for defining resource limits which act throughout the entire subhierarchy. A *cgroup* is therefore a collection of processes that share limits assigned to their cgroup and their ancestor cgroups in the hierarchy. While grouping of processes is implemented in the core cgroup kernel code, the

monitoring of resource usage is delegated to kernel components called *subsystems*, specific for resource type (cpu, memory, pids, rdma, freezer, etc.). The various kinds of subsystems currently implemented allow a fine-grained resource control, making it possible for example to limit the amount of CPU time and memory available to a cgroup, freezing and resuming the execution of processes in a cgroup and so on. For this reason subsystems are also called *resource controllers* [33]. This ability to assign resources to groups of processes and manage those assignments is fundamental for container-based virtualization and constitutes a powerful mechanism for avoiding Denial of Service (DoS) attacks, guaranteeing the availability of the assigned resources to all containers.

Capabilities are a list of privileges, traditionally associated to superuser, which can be independently enabled or disabled for each process [33], allowing a fine-grained permission checking.

Moreover, the Linux Security Modules (LSMs) enhance the access control on sensitive information, allowing to add another layer of protection to mitigate attacks that a container could perform against the host or other containers. Traditionally, access to resources is performed with ***Discretionary Access Control*** (DAC), which allows or denies the access by leveraging only on information about users and groups. *Yama* module extends the DAC support with additional system-wide security settings. Differently from DAC, the ***Mandatory Access Control*** (MAC) uses *policies* that specify authorization rules for allowing or denying access to sensitive resources that should be accessed in specific contexts: only if the requirements defined in the policy are satisfied, the access to the resource is gained. There are several implementations of the MAC concept considering different approaches, such as NSA Security-Enhanced Linux (*SELinux*), *AppArmor*, Simplified Mandatory Access Control Kernel (*SMACK*) and *TOMOYO* modules.

Taking advantage of these Linux kernel features, several containerization technologies have been implemented, based on two different approaches [1]:

- ***process containers*** create an isolated execution environment targeting a specific application which determines the container life cycle; so, the container lifetime starts when its target application begins to run and terminates when the target application finishes; *Docker* and *Rkt* are examples of this kind of containerization;
- ***machine containers*** create an isolated execution environment that targets multiple processes and services, customizable as traditional VMs; examples of this technology are *Linux Containers* (LXC) and (LXD); since they offer less flexibility, reusability and composability, they are less used in cloud environments than process container technologies.

More recently, a new approach has been proposed to implement lightweight virtualization addressed to cloud services, the ***Unikernels***. They combine strengths of process containers and VMs: like process containers, unikernels are single application oriented, like VMs they can be executed on hypervisors. The entire software stack of the application and its dependencies, comprising language runtime and system libraries, is compiled into a single bootable VM image that can be run directly on a standard hypervisor or the bare metal [34]. Differently from traditional VMs, they embed only what is needed to the application to run, so they require less system resources than VMs to run.

In the rest of the thesis work, we will focus on Docker containers which are currently the most used containerization technology in cloud environments, being integrated with all open source Linux based tools and taking advantage from the Kubernates orchestration system.

4.1.1 Docker

Docker is an open source platform for developing, deploying and running applications in *process containers*. All what is needed to an application to run in a Docker container is defined in an *image*, which can be considered a collection of files packaged together, containing all the "source code" of a Docker container. Images are made up of layered filesystems with different access privileges and layers ordered and stored in a single filesystem by using the Linux *Union filesystem*. The figure 4.2 represents the image hierarchy of Docker containers. The first layer

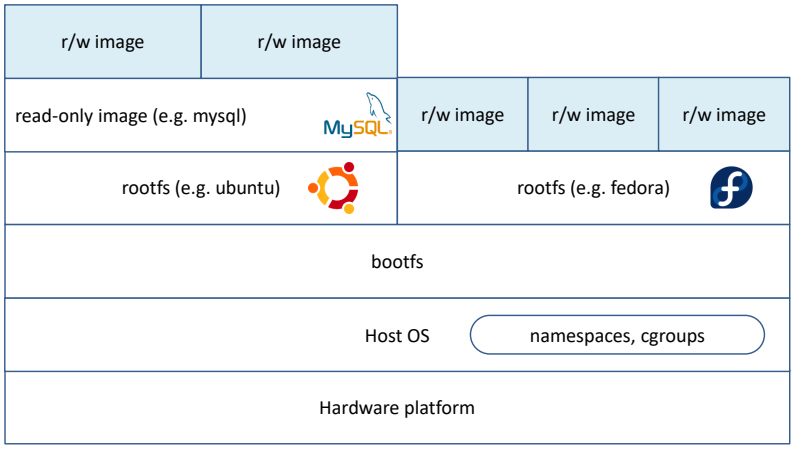

Figure 4.2. The Docker image hierarchy (source: [1])

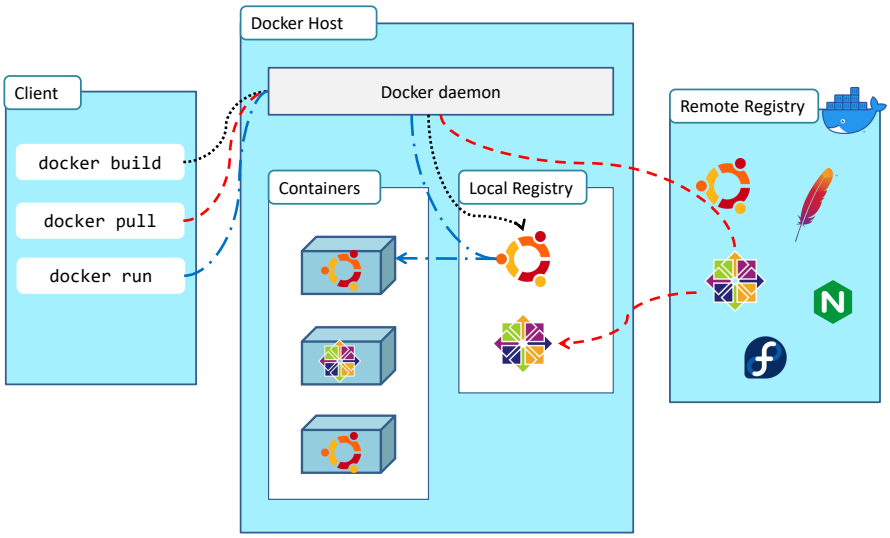

Figure 4.3. Docker architecture (source: [35])

of the hierarchy is the boot filesystem, *bootfs*, which contains data needed to boot the container. The second layer is the *rootfs*, or *base image*, which hosts the OS image. The upper layers contain application specific filesystems, such as the Apache web server, the MySQL server and so on. All these layers have read-only privileges since they can be shared among different containers; this optimizes the overall resource usage (disk and RAM) required by containers. On top of the read-only hierarchy, each container has a read-write layer containing the runtime information of a specific container. Docker manages all these layers by means of a *Storage Driver*, which by default is *overlay2* but can be configured with a different one, such as *aufs*, *devicemapper* or *btrfs*. Storage drivers perform the *copy-on-write* operations every time a container executes a writing operation on a read-only information, by copying it in its own read-write layer. We can create containers by using images already created and published in *registries*, or we can define a new image by writing it in a *Dockerfile*, a kind of script that defines the steps needed to create the image and run it, starting from a *base image* up to the target application.

Docker Architecture

The Docker platform uses a client-server architecture, as shown in fig. 4.3. The server-side is the *Docker daemon* (`dockerd`), which is the container runtime; its purpose is to manage Docker objects, such as images, containers, networks and volumes. The client-side can be:

- the `docker` *Command Line Interface* (CLI), which provides commands for performing operations on Docker objects;
- *Docker Compose*, a client specifically designed for managing applications composed of several containers.

Docker clients communicate with Docker daemons through a REST API, so they can run on the same system or on separated systems. The Docker platform provides a third component called *Registry*, which stores Docker images. When a Docker client asks a Docker daemon to pull or run an image, the daemon firstly searches the image in the local Registry; if the image is not found locally, the daemon looks for it on a configured remote Registry. *Docker Hub* is the public Registry configured by default in the Docker daemon, but there is the possibility to configure private Registries.

4.1.2 Major Risks for Core Components of Container Technologies

This section describes the major risks for the core components of container technologies (images, registries, orchestrators, containers, host OSs) on the basis of what is stated in the NIST specification 800-190 [36]. This analysis is applicable to most container deployments, regardless of the specific container technology or host OS.

Image Risks

A common risk in container-based environments is to have deployed containers with vulnerabilities, because generated from images containing components for which at some time new vulnerabilities have been discovered. This happens because, differently from traditional computing environments where the software deployed on the host is updated automatically, container updates must be performed first in the image files, and then these updated files have to be rebuilt by the container engine in order to free containers from new vulnerabilities. Moreover, even if all the image components are up-to-date, image configuration defects may expose the container to attacks. For example, an image may be erroneously configured to run as privileged user, giving to the container greater privileges than necessary; or, an image could contain an enabled SSH daemon, exposing the container to unnecessary risk of network-based attacks. Images could also refer to malicious files, included intentionally or inadvertently, which could be used to attack other containers or hosts within the environment. This happens especially when images are provided by unknown third parties, which can provide untrusted images. Another source of risk occurs when an application that uses secrets is packaged into an image. These secrets (such as username and password for accessing a database, private keys etc.) can be embedded directly into the image file system, letting anyone with access to the image to easily parse it and learn the secrets.

Registry Risks

Registry is a centralised entity to which container engines connect for downloading images to deploy in containers. This entity is crucial in a container architecture, so its inadequate protection or management can lead to several security risks. If the connection is performed over an insecure channel, it is subject to man-in-the-middle attacks for stealing developer or administrator credentials or secrets embedded in the image or for providing fraudolent or vulnerable images to the container engines. Moreover, it can happen that registries store stale images with known vulnerabilities. This does not represent a threat per se, but it increases the risk that the out-of-date version of the image is accidentally deployed in a container. Then, registries configured with weak authentication and authorization restrictions can lead to the compromise of the registry contents, or to the theft of proprietary images by attackers.

Orchestrator Risks

Administrators of an orchestrator typically have unbound access to the whole execution environment. However, an orchestrator may run applications managed by different teams, with different sensitivity levels; so the access to the orchestrator should be fine-tuned on the basis of specific needs, otherwise an administrator could inadvertitelly or intentionally corrupt the operation of some containers. Moreover, a poor management of orchestrator's accounts (e.g., if no longer needed accounts are not removed) can lead to unauthorized accesses and, since these accounts are highly privileged, this can expose to systemwide compromise. Another risk is represented by unauthorized access to container's data. Since the orchestrator manages the data storage volumes and these could contain sensitive data of containers' applications, it is necessary to encrypt data at rest to prevent unauthorized access.

Orchestrators are responsible to manage the virtual overlay networks that interconnect containers. A poor management of the inter-container interaction can lead to configure containers with different sensitivity levels in the same virtual network, exposing sensitive applications to greater risk from network attacks. Orchestrators are also responsible to manage the workloads of the hosts of the cloud environment. In the default configuration, orchestrators may place on the same host containers running applications with different sensitivity levels, thus exposing the most sensitive applications to a greater risk of attack.

Being the orchestrator the most important node of the cloud environment, its trustworthiness should be verified with particular care. A poor orchestrator configuration can lead to serious risks, such as unauthorized hosts joining the cluster, compromise of a single host implying the compromise of the entire cluster, unencrypted or unautehnticated communications between the orchestrator and other environment nodes.

Container Risks

Container runtime vulnerabilities can lead to scenarios in which software running in a container can escape from its sandbox and attack other containers or the host OS. Vulnerabilities can also be exploited by an attacker to alter the runtime software itself, allowing it to compromise other containers, monitor their communications and so on. Applications running in containers may themselves contain vulnerabilities that can lead the container to be compromised, allowing an attacker to access sensitive information or to attack other containers or the host OS.

Moreover, improper container runtime configurations can lower the stability and the security of the overall system. In particular, erroneous configuration settings may lead: to allow containers to invoke system calls that it would not be safe to call from inside containers; to run containers in privileged mode; to permit a container to mount sensitive directories on the host OS (such as `/boot` or `/etc`) so that it can change files in those directories, with negative consequences for the host and all the containers running on it.

Host OS Risks

The *attack surface* of a host OS is the collection of all the entry points ("attack vectors") which unauthorized users ("attackers") can exploit to enter data to or to extract data from the host OS. For example, any service accessible from the network provides a possible entry point to attackers, enlarging the OS's attack surface. The larger the attack surface is, more likely an attacker may find a vulnerability in an entry point in order to compromise the host OS and the containers running on it. Containers run on a shared kernel and this results in a larger inter-object attack surface than that relating to hypervisors. Moreover, since containers run inside the host OS, vulnerabilities in foundational system components (like cryptographic libraries or kernel primitives for process management) impact not only the host OS but also the containers running on it.

If administrators directly log on to host OSs for performing container management, they expose the platform to wide-ranging changes that could affect all containers running on the host, when they most likely wanted to manage containers of a specific application. Hence container management should happen through an orchestration layer.

4.1.3 Integrity verification of container-based environments

In order to contrast the issues previously exposed, an integrity verification mechanism should be in place for guaranteeing the trustworthiness of the software and the configuration data for all the entities involved in the container execution: the container runtime image, the container engine, the underlying host OS.

Docker, starting from version 1.8, natively offers *Docker Content Trust* (DCT) for verifying the container image against the creator's digital signature. A similar solution, called *Trusted Docker Containers*, is proposed by Intel for ensuring that images are not tampered before launching the container. These solutions however are not enough since they do not ensure the container integrity for its whole lifetime but at load-time only, while container image or configuration may be changed during runtime. Hence the need to develop solutions that allow to extend the RA mechanism to the attestation of container-based enviornments.

4.2 Docker Integrity Verification Engine (DIVE)

DIVE [1] is a solution proposed by the TORSEC research group of Polytechnic of Turin and, as the name suggests, it targets the Docker container engine. DIVE enables the integrity verification of host OS, container engine and containers running on a platform equipped with a TPM. This solution allows to identify, inside the IMA ML, the entries corresponding to the various running containers, thus giving the possibility to distinguish which container is compromised. In this way, the untrusted container can be immediately stopped and replaced by a freshly created one, without the need to reset the whole platform. The DIVE architecture comprises three components:

- the ***Attester*** is the target of the RA process; it represents a host machine belonging to the host cluster in the cloud environment, equipped with all the features that a Trusted Platform should have, that is a TPM and a root of trust for measurements that extends up to IMA; the Attester runs the Docker container engine and an ***RA Agent***, which is responsible to respond to attestation requests by sending Integrity Reports;
- the ***Verifier*** is the node responsible of verifying the integrity state of each Attester and of containers running on them;
- the ***Infrastructure Manager*** is the node in charge of creating the containers at user needs and keeping track of the mapping between the Attesters and the containers, identifying these entities with their *Universal Unique Identifiers* (UUIDs).

The Remote Attestation work-flow is shown in fig. 4.4. The Infrastructure Manager is in charge of periodically asking the Verifier to evaluate the trustworthiness of a list of containers, along with the list of Attesters on which the containers are running. To minimize the attack surface of the Attester's OS, the RA Agent does not accept incoming requests from third parties, rather it periodically polls the Verifier at a predefined time interval (1). If the RA Agent finds an RA request, it issues a quote command to the TPM for getting the PCR values signed with the TPM's AK (2), then it retrieves the IMA ML (3), creates the Integrity Report (IR) and sends it to the Verifier (4). When the Verifier receives the IR, it checks the integrity and the authenticity of the attestation information (5): firstly it verifies that the quote is valid, checking its digital signature against the public-key contained in the Attestation certificate stored in the Verifier when the Attester was registered; secondly, it verifies that the ML is not tampered with, by recalculating the extend operation on the template-hash of each entry and comparing the result against the IMA PCR contained in the quote. After having established the validity of the ML, only the entries of interest for the current attestation request are evaluated, retaining the entries belonging to the host system and to all the containers specified in the attestation request from the Infrastructure Manager. The measures contained in these filtered entries are then checked one by one against the reference database containing the whitelist (6). Finally, the Verifier returns the result of the integrity verification to the Infrastructure Manager specifying, in case of untrusted result, which measures are unknown from the reference database and which container they belong

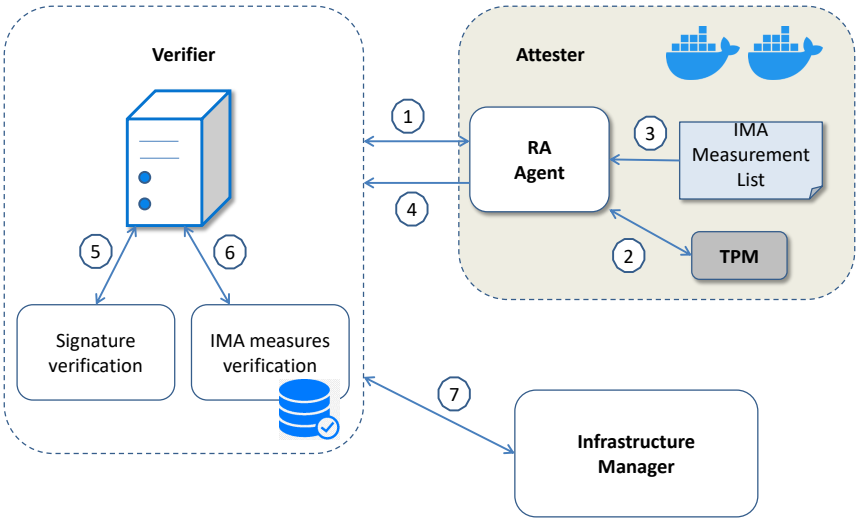

Figure 4.4. DIVE Remote Attestation work-flow (source: [1])

to (7). If the Verifier reported any compromised entity, the Infrastructure Manager can activate a roll-back strategy to restore the integrity state in the environment. For example, if a container results compromised, the Infrastructure Manager just needs to terminate the untrusted container and to start a new one, without rebooting the whole system to restore trust. Instead, if the host OS is compromised, then it is necessary to reboot the whole system otherwise the containers running on it are exposed to great risks (as discussed in section 4.1.2). The possibility to identify an untrusted container makes the RA process much more efficient in containerized environments, making it applicable in real scenarios.

The software prototype implemented for the DIVE architecture relies on *OpenAttestation* (OAT) SDK v1.7, a RA framework compliant with TCG specifications until version 1.7. OAT provides a *HostAgent*, corresponding to the RA Agent running on the Attester system, and an *Attestation Server* corresponding to the Verifier. These components expose a set of RESTful APIs for easy integration in other tools. OAT provides also a web interface to present the history of integrity reports. All communications within the RA workflow are encrypted relying on *Transport Layer Security* (TLS). In particular, communications between the HostAgent and the Attestation Server are performed on server-authenticated TLS connections, while communications between the Attestation Server and the Infrastructure Manager happen on mutual TLS connections, so that only authorized entities can request the trust level of the infrastructure to the Attestation Server. For what concerns the Infrastructure Manager, it could be a container management engine, such as Kubernates, or an orchestration platform such as OpenStack, that provides tools for supporting Docker containers in a cloud environment.

4.2.1 Attester prototype

The Attester, in the prototype implementation, relies on the following elements:

- the Linux IMA module, patched for supporting a new template called `ima-cont-id`;
- the TPM 1.2 device;
- the Docker container engine configured for using *Device Mapper* as storage driver;
- the OAT HostAgent, modified for supporting the DIVE architecture;
- a new command added to the Docker CLI client.

```
PCR#  template-hash     template    dev-id     filedata-hash   filename-hint

10    ccd75...21c04    ima-cont-id   8:19      sha1:1bc28...2c  /usr/bin/ls
10    74af2...1f412    ima-cont-id   8:19      sha1:123ac...31  /usr/sbin/sshd
10    aa1c1...89a2a    ima-cont-id   253:1     sha1:12cc5...d2  /usr/bin/ls
10    ff122...11b2a    ima-cont-id   253:1     sha1:762ad...aa  /badScript.sh
10    bc58a...739bf    ima-cont-id   253:2     sha1:8910f...ae  /usr/bin/find
```

Figure 4.5. IMA ML with `ima-cont-id` template (source: [1])

```
<Container Id="8948d6f37d41">
     <DevId>253:1</DevId>
</Container>
<Container Id="1beb7b9c05a6">
     <DevId>253:2</DevId>
</Container>
<Container Id="2f9695f9db36">
     <DevId>253:3</DevId>
</Container>
<Host>
     <DevId>8:0</DevId>
     <DevId>8:1</DevId>
     <DevId>8:19</DevId>
</Host>
```

Figure 4.6. Excerpt from the modified OAT Integrity Report (source: [1])

Processes running in Docker containers are seen by the host system as "normal" processes, so every time they load files into memory, these operations are automatically captured by the IMA module via the IMA Hooks present in the host system. Standard IMA templates neither allow to differentiate the MEs generated by container processes from those generated by other processes in the host system, nor permit to distinguish MEs belonging to one container from those belonging to another. So, the DIVE solution proposes the new IMA template `ima-cont-id`, that add to the ML entries an additional field, called `dev-id`, which correlates each file with its execution virtual device identifier, thus enabling the identification of the container that generated the ME. This feature is based on the Docker's *Device Mapper* storage driver, which assigns a different virtual device identifier to the processes running in each container. The figure 4.5 shows an example of IMA ML with the `ima-cont-id` template, where the first two entries belong to the host system, identified by `dev-id` 8:19, the subsequent two entries belong to a container identified with `dev-id` 253:1, the last entry belongs to another container identified with `dev-id` 253:2.

The modifications to the OAT HostAgent regard adding new information in the IR, used by the Verifier to map the device IDs stored in the ML entries to the container UUIDs. The figure 4.6 shows the new XML items added to the OAT Integrity Report: `<Container>`, that allows the mapping between a Docker container UUID and the virtual device ID associated to it by Device Mapper, and `<Host>`, that contains the list of all physical device IDs associated to the host system. Finally, the Docker *Command Line Interface* (CLI) has been extended with a new command, for retrieving the mapping between the container UUID and its device ID in an efficient way.

4.2.2 Verifier prototype

The Verifier prototype relies on the OAT Attestation Server, extended with a new type of analysis that doesn't consider the Attester as a whole, while it evaluates the integrity state of the entities specified by the Infrastructure Manager in the attestation request, that are a list of containers along with a list of Attesters on which the containers are running. The analysis ignores the

integrity state of containers that are not specified in the attestation request performed by the Infrastructure Manager.

The Verifier receives the full IMA ML and, after having verified its integrity, only the measurements contained in the entries of interest are compared with the those contained in a reference database. This database is initially populated with the names and digests of all the elements contained in packages stored in official repositories for Linux distributions. Since the `ima-cont-id` template relates each entry with the host or a specific container, the result of the verification process is able to specify which measures are unknown and which entity they belong to (container or host); only if the host system of the Attester results untrusted, the Infrastructure Manager have to reboot the entire platform, otherwise it can restart only the compromised container, while the rest of the platform is not interrupted.

4.2.3 Conclusions

DIVE is a solution for Docker container attestation that relies on the hardware RoT enabled by the TPM for the attestation information, making the IR non-forgeable. This strong protection against remote attacks comes at a nearly negligible performance impact on the Attesters, feature that makes this solution appliable in real scenarios. Another important feature of DIVE is that the Verifier is able to identify which container or host system is compromised, allowing the Infrastructure Manager to take the most efficient decision to restore trust: restarting only the compromised containers or the entire physical platform. Moreover, DIVE doesn't require any modification to the applications running in containers, the only modifications needed to enable this solution are applied to the host system. This makes DIVE very easy to be adopted in real case scenarios.

Limitations of DIVE regard the fact that it needs Device Mapper as storage driver for Docker, but there are scenarios in which other storage drivers provide better performance and stability. Another drawback is that collisions in the device Id assigned to containers may occur; in particular, if a container is stopped and a new one is started, the new container could acquire the same device Id of a previously terminated container; this means that the new container will inherit all the entries in the ML from the previous one, possibly resulting in an attestation fail when the integrity check is performed for the new container.

4.3 Container-IMA

Container-IMA [2] is another solution for container attestation, proposed by a research group of Peking University in China. This work addresses privacy issues that the RA can cause when containers belonging to different users run on the same physical host. Other solutions based on vTPM obtain a separation of the attestation information among the different containers, but they suffer from several security and efficiency issues. Container-IMA, as DIVE, is a solution based on IMA and it requires neither an additional layer in user space (like vTPM), nor any modification to the existing applications running in containers. The proposed implementation is based on the OAT framework and TPM 1.2.

The figure 4.7 shows a use case scenario, in which the *prover* (that is the attesting system) hosts two containers belonging to user A and two containers belonging to user B. The *verifier* represents a remote user that wants to know the integrity status of his containers running in the *prover*. The `container management services` represent the *container runtime engine* (such as Docker Daemon), which is responsible for starting and managing containers. All other processes running in the underlying host system are classified as `host applications`. Container-IMA assumes that the *prover* is equipped with a TPM (the prototype implementation supports TPM 1.2) and supports trusted boot for providing an integrity evidence of the firmware components and the OS kernel. The threat model for this solution does not cover physical attacks to the host system and doesn't detect runtime memory attacks, which can be mitigated by leveraging other mechanisms, such as address space layout randomization and control flow attestation. The adversary is classified in two categories: a *local adversary* is capable of eavesdropping on, and

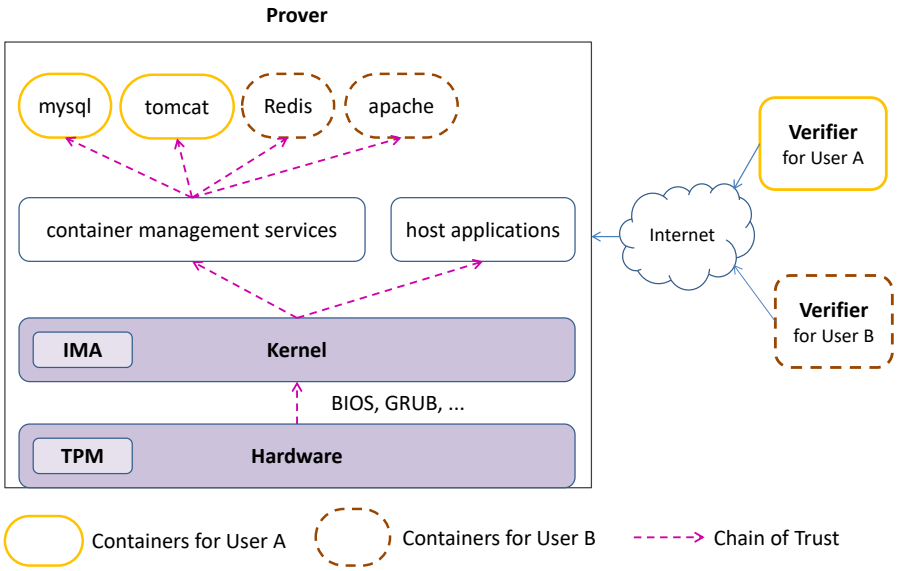

Figure 4.7. Use Case in a Container Setting (source: [2])

interfering with, the *prover*'s communication; a *remote adversary* can remotely infect the prover with malware, modifying files or integrity evidence related to containers, corrupting the attestation mechanism or impersonating as `container management services`. Finally, Container-IMA assumes that only authorized Verifiers can receive attestation information.

In a platform that runs containers, the chain of trust, established with trusted boot and extended to the OS kernel thanks to IMA, can be decomposed as follows [2]:

1. *Integrity of prover's Boot Time* (I_{boot}^{Pro}): starts from *prover* power on and ends with OS kernel loading; it includes BIOS, GRUB and OS kernel;
2. *Integrity of Containers' Dependencies* (I_{dep}^{Con}): refers to the `container management services` and all files and libraries required by them;
3. *Integrity of a Container's Boot Time* (I_{boot}^{Cont}): refers to the images and boot configurations used by `container management services` to launch a container;
4. *Integrity of a Container's Applications* (I_{app}^{Con}): starts when `container management services` launch a container and ends when the container terminates; it includes all processes and files belonging to a container;
5. *Integrity of Host Applications* (I_{app}^{Host}): starts when the OS kernel is successfully launched and ends when the *prover* is shut down. The `container management services` and the containers do not belong to this partition.

The figure 4.7 highlights the containers' chain of trust. Containers directly depend on the `container management service`; thanks to namespaces, containers are isolated from other `host applications`, so the chain of trust for a given container includes I_{boot}^{Pro}, I_{dep}^{Con}, I_{boot}^{Con} and I_{app}^{Con} (the last two belonging to the container to be attested). When a *verifier* requests the attestation information for a container, the *prover* can aggregate only these subsets of information, while other measurements not belonging to these partitions should not be revealed to the *verifier*, in particular information belonging to I_{app}^{Host} and other containers' I_{app}^{Con} and I_{boot}^{Con}. Moreover, the *verifier* cannot distinguish whether his container is the only one running on the *prover* or not. To realize this privacy requirement, Container-IMA subdivides the traditional IMA ML into the partitions

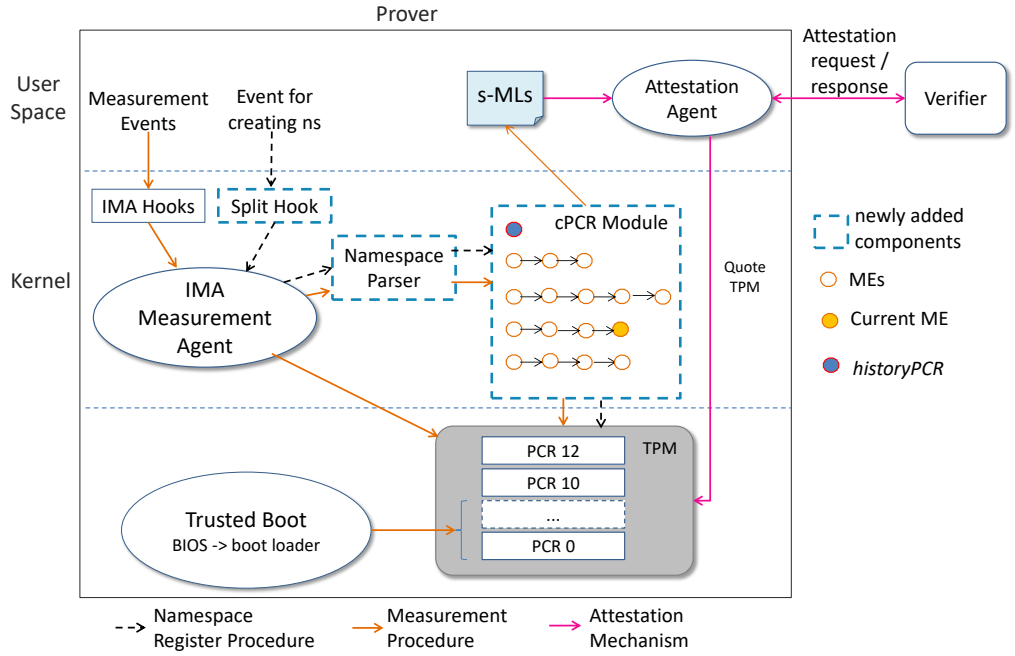

Figure 4.8. Overview Architecture of Container-IMA (source: [2])

mentioned above. In order to realize the ML partitioning, Container-IMA modifies the IMA module by adding three new modules, as shown in figure 4.8: the *Split Hook*, the *Namespace Parser* and the *cPCR Module*. The overall architecture that allows individual container attestation is based on two mechanisms: the *measurement mechanism* and the *attestation mechanism*.

4.3.1 Measurement Mechanism

The measurement mechanism consists of two phases, the *Namespace Register Procedure*, which allows the initial partitioning of the ML belonging to a container on basis of its mount namespace, and the *Measurement Procedure*, that allows to add a given ME to its corresponding ML partition. As previously mentioned, the chain of trust of a container is composed of the following integrity measurements: I_{boot}^{Pro}, I_{dep}^{Con}, the container's I_{boot}^{Con} and I_{app}^{Con}. The integrity of the *prover* at boot time, the I_{boot}^{Pro}, is measured by the trusted boot and protected by PCR0-7; the MEs generated in the boot phase are already in a separate ML, so they are not considered in the following description.

Namespace Register Procedure

As described in paragraph 4.1, processes running in a container are isolated from those belonging to other contaienrs and the host system by means of namespaces. This is the feature used by Container-IMA to distinguish the MEs generated by a container from those generated by other containers and from the host applications. By parsing the mount namespace number of the process that spawned the event, the I_{app}^{Host} and the various I_{app}^{Con} can be separated.

The workflow of the Namespace Register Procedure is depicted in figure 4.8. The Split Hook, introduced by Container-IMA, is responsible to notify the IMA Measurement Agent that the system call `unshare()` has been invoked to generate a new mount namespace. As response to this event, the IMA Measurement Agent has to allocate new data structures so that the MEs generated by the processes in this new namespace can be separated by the others. For each namespace Container-IMA maintains in the kernel memory a measurement list, each of them having its corresponding *separated Measurement Log* (s-ML). So, a s-ML contains the MEs generated in a

given namespace *ns*:

$$s\text{-}ML =< \{measure(ME_{ns})\}, ns > \tag{4.1}$$

If n is the number of namespaces generated through `unshare()`, the set of all s-MLs can be represented as:

$$s\text{-}MLs = \{< \{measure(ME_{ns})\}, ns >\}_n \tag{4.2}$$

Container-IMA provides a hardware-based RoT to the ML's partitions, but it can not accomplish this by assigning a different PCR to the various partitions, since it would be unfeasible for tens or hundreds of containers running on a host system. So, Container-IMA implements the *container-based PCRs (cPCRs) Module*, responsible to manage the cPCRs data structure in the kernel memory. Each cPCR has three data associated to it:

1. *value*, that contains the digest resulting from the extentions of the MEs belonging to a given namespace;
2. *ns*, that is the namespace corresponding to this cPCR;
3. *secret*, which is a random number generated by the TPM, used to hidden the cPCR *value* to *verifiers* that have to attest their own containers but do not have the ownership of the container corresponding to *ns*.

If n is the number of ML's partitions, the list of all cPCRs can be represented as:

$$cPCR\text{-}list = \{cPCR\}_n = \{< value, ns, secret >\}_n \tag{4.3}$$

When the IMA Measurement Agent receives from the Split Hook the event for creating a new namespace, it notifies the Namespace Parser to parse the new mount namespace number and pass it to the cPCR Module. This firstly requests to the TPM to generate a new random number to use as secret, then it creates a new cPCR and a new s-ML for the namespace just created:

$$cPCR\text{-}list_{new} := cPCR\text{-}list_{old} \cup \{< all\text{-}zero, ns, secret >\} \tag{4.4}$$

$$s\text{-}MLs_{new} := s\text{-}MLs_{old} \cup \{< \{\}, ns >\} \tag{4.5}$$

Measurement Procedure

The Measurement Procedure is responsible for measuring and storing MEs. When a ME is generated, an IMA Hook notifies it to the IMA Measurement Agent, which measures this ME if it matches the IMA Policy. Then the Namespace Parser receives the ME and tries to retrieve its mount namespace number. If the ME does not have a mount namespace or its mount namespace has not been previously registered in the cPCR Module, this ME is considered belonging to the `host applications` and managed with traditional IMA behaviour, so the measurement result is extended in PCR10 and recorded in the `ascii_runtime_measurements` file. Otherwise, if the ME has a mount namespace previously registered through the Namespace Register Procedure, the Namespace Parser passes it to the cPCR Module, which locates the cPCR corresponding to its mount namespace (*target-cPCR*) and extends the ME template-hash in the *target-cPCR.value*, by performing the same *extend* operation defined in the TPM specifications for PCRs:

$$target\text{-}cPCR.value_{new} := HASH(target\text{-}cPCR.value_{old} \,||\, ME.template\text{-}hash) \tag{4.6}$$

Then, the cPCR Module appends the ME in the target s-ML. In this way, the integrity of each s-ML is protected by its corresponding cPCR.value and a *verifier* can check that the received s-ML is not tampered with by comparing the cPCR.value with the result of the *extend* operation computed over the template-hashes of each s-ML entry. However, the cPCRs are in the kernel memory, so the s-MLs integrity is not protected by a hardware-based RoT.

In order to provide a hardware-based RoT to the s-MLs, the cPCR Module binds the cPCRs into a physical PCR that is not already used for other purposes; PCR12 is the default PCR for

extending cPCRs in the prototype implementation, but Container-IMA provides a kernel command line parameter to change the PCR index. Before proceeding with the extension, the cPCR Module records the current value of PCR12 in a internal variable called *historyPCR*, which will be sent to the *verifiers* for allowing them to check that the cPCR list has not been tampered with. After that, the cPCR Module computes, for each $cPCR_i$, its $sendcPCR_i$ value that corresponds to the $cPCR_i.value$ xored with the $cPCR_i.secret$:

$$sendcPCR_i := cPCR_i.value \ \ xor \ \ cPCR_i.secret \tag{4.7}$$

This operation is performed because, when a *verifier* receives the list of the sendcPCRs, it can disclosure the cPCR.values only for those containers of which it knows the secrets, while the other cPCR.values remain hidden, preventing a malicious *verifier* from getting ideas of what software is running inside other containers. The *verifier* that owns a container should be notified about its corresponding *secret* when the container is created; after container creation, the kernel does not provide the *secret* anymore. Then, all the sendcPCRs are extended one after the other, simulating the *extend* operation performed by the TPM:

$$tempPCR_0 := sendcPCR_0 \tag{4.8}$$

$$tempPCR_i := HASH(tempPCR_{i\text{-}1} \,||\, sendcPCR_i),\ \forall i \in [1, n] \tag{4.9}$$

Finally, the resulting $tempPCR_n$ value is extended in the TPM's PCR12, where $PCR12_{old}$ has been recorded into the *historyPCR* variable:

$$PCR12_{new} := PCR\_Extend(PCR12_{old},\ tempPCR_n) \tag{4.10}$$

In this way a hardware-based RoT is established, because TPM's PCR12 protects the integrity of the sendcPCRs list and each sendcPCR protects the integrity of its corresponding s-ML. In particular, when a *verifier* wants to check the integrity of a given container, for example container A, it has to receive the container's $s\text{-}ML_A$ (containing all the MEs of container A), along with the *historyPCR* value, the list of all sendcPCRs and the quote containing the PCR12 value signed with the TPM's AK. Firstly, the *verifier* checks that the sendcPCRs list is genuine and, in order to do that, it recomputes the extend operation on all the sendcPCRs as explained by equations 4.8 and 4.9; then, the final $tempPCR_n$ value is extended with the *historyPCR* value:

$$PCR12 \ = HASH(historyPCR \,||\, tempPCR_n) \tag{4.11}$$

If the resulting value is equal to the signed PCR12 value received with the quote, then the sendcPCRs list is authentic. Now, if the *verifier* knows the $secret_A$ corresponding to container A, the *verifier* uses it to disclosure the corresponding $cPCR_A.value$:

$$cPCR_A.value = sendcPCR_A \ \ xor \ \ secret_A \tag{4.12}$$

Finally, the *verifier* can use the $cPCR_A.value$ to check the integrity of $s\text{-}ML_A$ by extending all the template-hashes contained in each entry and checking that the resulting value matches $cPCR_A.value$.

The process described until now allows to partition the I_{app}^{Host} and the I_{app}^{Con}s, but it doesn't allow to record in a separate ML the *Integrity of Containers' dependences* (I_{dep}^{Con}) and *Integrity of Containers' Boot Time* (I_{boot}^{Con}). Regarding I_{dep}^{Con}, the idea is to reuse the Namespace Register Procedure also for partitioning the Containers' dependencies but, since the `container management service` does not run in a new namespace, in order the solution to work, it is necessary to launch the containers' dependencies through a program, named `bootstrap program`, whose purpose is to create a new namespace in which the containers' dependencies will run. In the implemented prototype, the researchers chose `/usr/bin/unshare` as the `bootstrap program`. In this way, when the `container management service` will be launched through `/usr/bin/unshare`, the Split Hook will be triggered and a new cPCR and s-ML will be allocated for it. Since the `container management service` is launched before any other container, it will always correspond to $cPCR_0$. Moreover, since all *verifiers* will need the containers' dependencies secret, the $cPCR_0.secret$ is set to a well known value, that is all-zero. Obviously, for this solution to work,

```
==> 4026532222 <==

4026532222  ccd75...21  ima-ng sha1:38919a... 1990->1980->1907->1447->1249->1071->1
                                                                    ->0_4026532222:/usr/bin/unshare
4026532222  74af2...1f  ima-ng sha1:a348d3... 4026532222:/usr/bin/dockerd
4026532222  aa1c1...89  ima-ng sha1:80a5ea... 4026532222:/usr/bin/docker-containerd
4026532222  ff122...11  ima-ng sha1:126ee5... 4026532222:/var/lib/docker/tmp/docker-default618280113
4026532222  bc58a...73  ima-ng sha1:a00d30... 4026532222:/lib/modules/3.13.11-
                                                                  ckt39/kernel/ubuntu/aufs/aufs.ko

==> 4026532238 <==

4026532238  2a34c..14  ima-ng sha1:d5442f... 2358->2354->2346->2001->1990->1980->1907->1447->1249->1071
                                                              ->1->0_4026532238:/usr/local/sbin/runc
4026532238  f1aee..ac  ima-ng sha1:611a59... 4026532238:/bin/bash
4026532238  78fcb..1d  ima-ng sha1:b43aec... 4026532238:/lib/x86_64-linux-gnu/ld-2.23.so
4026532238  45cdd..df  ima-ng sha1:eaae87... 4026532238:/etc/ld.so.cache
4026532238  ace7e..5b  ima-ng sha1:2bd938... 4026532238:/lib/x86_64-linux-gnu/libtinfo.so.5.9


==> ascii_runtime_measurements <==

10  17f4a..21  ima-ng sha1:27d6a1... boot_aggregate
10  accb7..3a  ima-ng sha1:a5e65f... 4026531840:/init
10  09acc..83  ima-ng sha1:dc3e62... 4026531840:/bin/sh
10  cbefe..a6  ima-ng sha1:67c253... 4026531840:/lib64/ld-linuc-x86-64.so.2
10  34dda..c1  ima-ng sha1:fac553... 4026531840:/etc/ld.so.cache


==> docker-boot <==

"... [4026532238] sha256:7aa3602ab41e...  ...
    /var/lib/docker/containers/.../config.v2.json" c952b062e8be3cbc407242cb2ebcb27c8111b489
```

Figure 4.9. An example of the first 5 entries of each ML partition after having bootstrapped `dockerd` through `unshare` command and then set up a new container `docker run -it ubuntu:16:04` (source: [2])

the first namespace created with the unshare syscall in the system must be that created for the containers' dependencies. So, when a *Verifier* attests a container, it has to receive the $s\text{-}ML_0$ along with the attestation information previously mentioned.

There is another problem to be solved: how can a *verifier* prove that its container has been actually launched through the containers' dependencies whose s-ML it received? Container-IMA solves this issue by adding as first entry in the s-MLs the measurement of the process that created the new namespace (`createProcess`), adding in the template of the first entry the process PID and the PIDs of all its ancestors. So, the first s-ML's entry is different from other entries, firstly becacuse it does not represent a ME that matches the IMA Policy but represents the "creation of a new namespace" event, secondly because it has a different template. In order to prove that its container has been launched through the containers' dependencies whose measures are stored in $s\text{-}ML_0$, a *verifier* checks the PIDs: if the PID-chain stored in the container's s-ML contains the PID of the `bootstrap process` (`/usr/bin/unshare` in this case), that is the first PID in the PID-chain contained in $s\text{-}ML_0$, then the measures contained in the $s\text{-}ML_0$ are indeed those generated by the dependencies of its container, otherwise they are not. The figure 4.9 shows an example of ML partitions. The s-ML files are named with the mount namespace number they represent, in particular file `4026532222` refers to the s-ML for containers' dependencies (I_{dep}^{Con}) and file `4026452238` refers to the s-ML for a container (I_{app}^{Con}); while file `ascii_runtime_measurements` contains measurements related to the host applications (I_{app}^{Host}), which are deemed irrilevant for the containers attestation by Container-IMA developers. The template used in the prototype implementation is `ima-ng`, with some fields modified from the original template:

1. `index`, the namespace number or the PCR index (in the case of `ascii_runtime_measurements`);
2. `template-hash`, the hash that protects the entry information;

3. `template-name`;
4. `file-hash`, the hash computed over the file data;
5. `file-path`, the file absolute path, with a prefix equals to the mount namespace number and, for the first entry only, the PID-chain.

In file `4026532222` the `createProcess` is `/usr/bin/unshare`, whose PID is 1990, while in file `4026532238` the `createProcess` is `/usr/local/sbin/runc`, whose PID is 2358 and has the process with PID 1990 among its ancestors. It follows that file `4026532222` actually represents the dependencies for the container represented by file `4026532238`.

The *Integrity of Containers' Boot Time* (I_{boot}^{Con}) comprises the image and configurations to bootstrap a container. In order to store the measurements on these information in a separate ML, Container-IMA extends the chain of trust from the IMA module to the containers' dependencies; that is, since containers' dependencies have been measured by IMA and the measurements are protected by a hardware-based RoT, the containers' dependencies could implement the function of *Measurement Agent* with respect to the containers' boot time information. In the prototype, the researchers implemented the Measurement Agent in the `runc` process, which collects the measurements on the containers' image and configurations in a file called `docker-boot` and extends the template-hash computed on each entry of this file in the physical PCR11. The figure 4.9 shows an example of `docker-boot` file, whose template has six fields:

1. `id`, the Container ID;
2. `ns`, the mount namespace number;
3. `HASH(image)`, the digest computed over the container's image;
4. `HASH(config)`, the digest computed over the container's configuration;
5. `PATH(config)`, the absolute path of the container's configuration file;
6. `template-hash`, the digest computed over the entry's information.

For privacy reasons, the `docker-boot` file is not transmetted as is to the *verifiers*: the entry related to the container owned by the *verifier* is transmitted with all its information, while, for the other entries, only the `template-hash` is transmitted. This lets the *verifier* to check the integrity of the container's boot information against the PCR11 value, at the same time not revealing other containers' information.

4.3.2 Attestation Mechanism

The Container-IMA Attestation Mechanism enables a *verifier* to attest the integrity of a given container and its dependencies. Container-IMA assumes that an effective user management system that identifies the *verifiers* is already in place, such as Kubernates, preventing unauthorized *verifiers* from receiveing attestation information.

Message Transferring

When an authorized *verifier* wants to attest the integrity of a container running in a *prover*, it sends to the *Attestation Agent* a `request:<nonce, containerID>`, where *nonce* is a random number generated by the *verifier*, and *containerID* is the UUID of the target container. When receiving the request, the *Attestation Agent* sends to the *verifier* an IR containing the following information:

1. a TPM's *quote*, that is $Sign_{AK}\{nonce \,||\, PCRs\}$, where the PCRs are PCR0-7 (for the *prover*'s boot time), PCR11 (for containers' boot time measures) and PCR12 (for containers' dependencies and applications);

2. *sendcPCRs* and the *historyPCR* (that is the old value of PCR 12);
3. s-ML for *prover*'s boot time (I_{boot}^{Pro});
4. s-ML for container's dependencies (I_{dep}^{Con});
5. s-ML for container's boot time (I_{boot}^{Con}), containing for the target container the complete information, while for other containers only the template-hash;
6. s-ML for the target container's applications (I_{app}^{Con}).

Verifier Workflow

When receiving the response from the *Attestation Agent*, the *verifier* firstly validates the authenticity of the TPM quote by checking the signature with the public part of the AK. If the quote is authentic, the *verifier* validates its freshness by checking if the *nonce* is the one sent in the IR request.

If these checks are passed, the *verifier* uses the trusted PCR values for checking the integrity of all the received s-MLs. In particular, the integrity of the s-MLs for *prover*'s boot time (I_{boot}^{Pro}) and the container's boot time (I_{boot}^{Con}) can be easily checked by simulating the `PCR_Extend` operation and comparing the result with the corresponding trusted PCRs (PCR0-7 for I_{boot}^{Pro} and PCR11 for I_{boot}^{Con}). While, for checking the integrity of the s-MLs for I_{dep}^{Con} and I_{app}^{Con}, the *verifier* has first to check the validity of the received *sendcPCRs*, so it performs the extend operation with all $sendcPCR_i$ for obtaining *tempPCR*, as shown in equations 4.8 and 4.9; then it extends *tempPCR* with the *historyPCR* and verifies if the final value is equal to the trusted PCR12. If the values match, the *verifier* uses the contaienr's *secret* for getting its cPCR.value and *all-zero* for getting $cPCR_0.value$, then it uses these values for verifying the integrity of the s-MLs for container's applications (I_{app}^{Con}) and container's dependencies (I_{dep}^{Con}), respectively. Finally, the *verifier* checks whether the received I_{dep}^{Con} is actually its container's dependency by checking the PID-chain of the container's `createProcess`. If all these verifications are passed, the *verifier* can use the s-MLs for evaluating the trustworthiness of the container, by comparing the measurements contained in them against its expectations, which are collected from software and hardware manufacturers.

Chapter 5

Keylime Framework Analysis

Keylime born out of the security research team in MIT's "Lincoln Laboratory" and was presented to the scientific community, in December 2016, with the whitepaper "Bootstrapping and Maintaining Trust in the Cloud" [37]. It is currently a "Cloud Native Computing Foundation" (CNCF) hosted project which provides an open source solution both for bootstrapping hardware rooted cryptographic identities for cloud nodes and for system integrity monitoring of those nodes via periodic attestation.

5.1 Background

Keylime's idea starts from the observation that IaaS services are becoming more and more popular among companies. "Infrastructure as a Service" (IaaS) is a cloud computing service model where fundamental computing resources (processing, storage, networks) are provisioned to customers to deploy and run arbitrary software, which include operating systems and applications. These IaaS resources are referred to as *cloud nodes* and can be physical hardware, virtual machines or containers. With this service model, *cloud tenants* do not manage or control the underlying cloud infrastructure but control operating systems, storage and deployed applications. Typically, customers are able to self-provision this infrastructure by using a web-based graphical user interface that acts as a IT operations management console for the overall environment. This console allows customers to upload a whole image to the provider infrastructure or to configure a pared-down base image made available by the provider. Users often start by customizing a provider-supplied image and then, when they achieved the desired configuration, they create their own image by using tools like `Packer`, to speed up the deployment of new cloud nodes. IaaS is nowadays very popular among companies since it offers advanced technologies with savings of time and money, allowing them to:

- access applications and data, from anywhere and at any time, simply using a PC connected to Internet;
- avoid the initial expense of setting up and managing a datacenter;
- implement innovation quickly, because the computing infrastructure necessary to launch a new product or a new initiative can be ready in a matter of hours, versus the days, weeks or even months needed to set up a local datacenter;
- focus on the main business of the company, without wasting human resources for managing IT infrastructures.

However, IaaS cloud service providers do not currently offer all the required components to establish a trusted environment for hosting sensitive data and business critical applications. Tenants have limited ability to verify the underlying platform when they deploy their application to the

cloud and to verify that the platform remains trusted for the entire duration of their computation. Tenants have also a limited ability to establish unique unforgeable cryptographic identities, bound to a hardware RoT, in the cloud nodes. Frequently, those cryptographic identities rely exclusively on a software-based solution and often the methods used to inject the identities in the cloud nodes require trust in the service provider, since secrets are passed unprotected to the cloud nodes via the cloud provider. Typically, cloud providers use `cloud-init` as a standard mechanism for allowing tenants to specify bootstrapping data, including secrets, for the cloud nodes. These bootstrapping data are not encrypted, thus allowing a provider to intercept them.

A solution for bootstrapping trust and detecting changes in the system state is represented by the TPM but it has not been widely used in IaaS cloud environments for the complexity of its standards, the difficulty of their implementation, the low performance (500+ms to generate a digital signature), the fact that the TPM is a physical device while most IaaS services rely upon virtualization, which separates cloud nodes from the underlying hardware on which they run. Although the Xen hypervisor supports a virtualized TPM that has a hardware RoT in the physical TPM, IaaS environments don't have protocols that allow to use vTPMs.

In order to address these issues, Keylime developers identified a set of desirable features that an IaaS trusted computing system should have:

- ***Secure Bootstrapping***: the system should allow a tenant to securely inject an initial root secret into each one of his cloud nodes; then, the tenant can use this initial secret to chain other secrets, so enabling higher level security services;
- ***System Integrity Monitoring***: the system should enable the tenant to monitor the integrity state of cloud nodes and detect integrity deviations, reacting within one second;
- ***Secure Layering (Virtualization Support)***: the system should enable a tenant for secure bootstrapping and integrity monitoring also in VMs by leveraging a TPM in the provider's infrastructure; this requires provider collaboration, but it must be done giving the provider the least privilege;
- ***Compatibility***: the system should permit the tenant to use hardware-rooted cryptographic keys in software so that the services they already use are made more secure (such as disk encryption or configuration management);
- ***Scalability***: the system should scale to support secure bootstrapping and integrity monitoring of thousands of cloud nodes, since IaaS resources can be elastically spawned and deleted.

Keylime is proposed by MIT's researchers as the first end-to-end IaaS trusted cloud key management service that supports all the desirable features listed above:

- it implements a new bootstrap key derivation protocol for injecting identities and other secrets into cloud nodes, combining both tenant intent and integrity measurements (*Secure Bootstrapping*);
- it performs periodic remote attestations, linking the identity revocation of a cloud node to integrity deviations of the system (*System Integrity Monitoring*);
- it provides the previous functionalities in both bare-metal and VMs in a way that minimizes trust in the cloud provider (*Secure Layering*);
- it has been integrated with applications and services common to IaaS cloud deployments and non-trusted-computing aware, such as `cloud-init`, IPsec, Puppet, Vault, LUKS (*Compatibility*);
- it can handle thousands of cloud nodes simultaneously, managing to check thousands of IRs per second (*Scalability*).

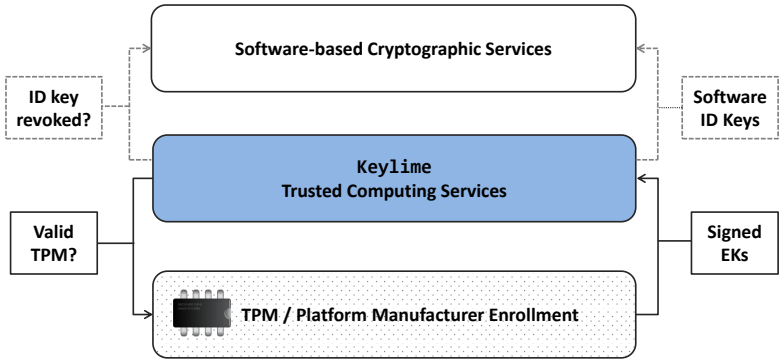

Figure 5.1. Decoupling of trusted hardware from high-level security services (source: [37]).

The central idea of Keylime was to integrate Trusted Computing with high-level security services, leveraging on integrity measurements for bootstrapping and revoking identity in the cloud nodes, allowing higher-level services that use these identities (IPsec, Puppet, etc.) to work independently, without the need to be trusted computing aware. As shown in figure 5.1, Keylime provides a software layer between trusted hardware (TPM) and software-based security services, exposing a clean and easy to use interface that allows integration with existing security technologies.

5.2 Design

Keylime has been proposed as a solution to overcome security issues that arise in cloud computing environments, as it is capable to provide a hardware root of trust to tenants in order to establish the trustworthiness of the IaaS infrastructure and of their own systems running on that infrastructure.

5.2.1 Threat Model

The threat model of Keylime assumes that the cloud provider is "semitrusted", that is, it is organized in a trustworthy manner but still susceptible to compromises or malicious insiders. In particular, Keylime assumes that [37]:

- the cloud provider has all the processes, technical controls and policies for limiting the impact of compromises, preventing them from spreading throughout the entire infrastructure, but a fraction of the provider's resources can be under the control of the adversary, for example a subset of racks in an IaaS region may be controlled by a rogue system administrator;
- the adversary can arbitrarily monitor or manipulate compromised portions of the cloud network or storage;
- the adversary cannot physically tamper with host resources, such as CPU, bus, memory or TPM;
- the provider does not intentionally deploy a hypervisor coded with the explicit purpose of spying on tenant VM memory;
- the TPM is provided with a certified EK, which establishes the authenticity of the TPM hardware;
- the adversary's goal is to obtain persistent access to tenant resources in order to steal, disrupt or deny the tenant's data and services, and this can be done modifying the code at load-time or the process at run-time;

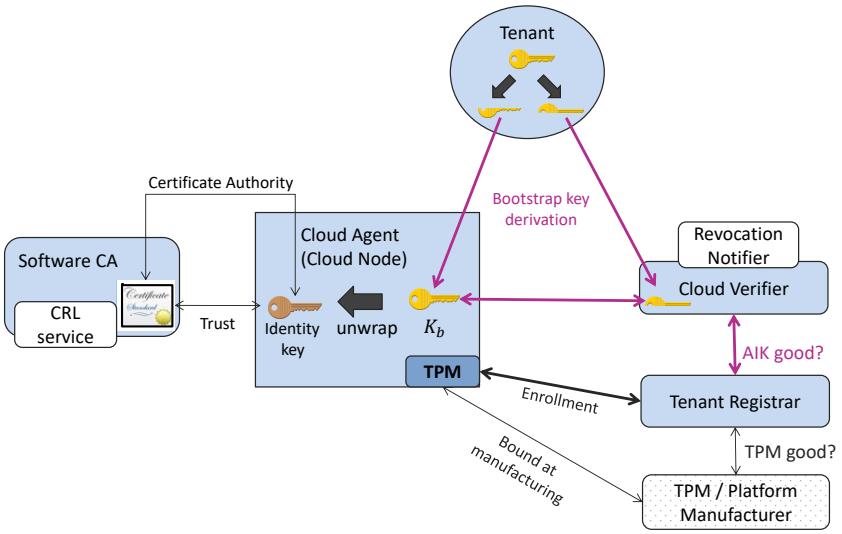

Figure 5.2. Keylime simplified architecture.

- load-time modifications can be detected by integrity measurements of the hypervisor/kernel (with trusted boot) and of the applications (with IMA);
- run-time modifications can be detected by runtime integrity measurements of the kernel (LKIM [38]) and of the applications (DynIMA [39]).

5.2.2 Simplified Architecture for Physical Nodes

This section describes the simplified architecture for managing trusted computing services when cloud nodes are physical hosts. The simplified architecture, represented in figure 5.2, comprises the following components.

The ***Registrar*** stores valid AIK_{pub} keys, EK_{pub} keys and EK certificates and indexes them by means of cloud nodes' UUIDs. The *registrar* is a simplified implementation of the TCG *Attestation CA* because it implements the protocol for the Attestation Key Identity Certification [40], proving that the node which holds the EK_{priv} key knows also the AIK_{priv} key, but it doesn't validate the EK certificate (this validation is performed by the Tenant during the key derivation protocol, described in section 5.2.2) and doesn't hide the EK_{pub} key to its clients. Clients request TPM's credentials from the *registrar* through a server authenticated TLS channel.

The ***Cloud Verifier (CV)*** is the core component of the Keylime architecture since it is responsible for verifying the integrity state of the tenant's IaaS resources. The *CV* relies upon the *registrar* for retrieving the AIK_{pub} key needed for the validation of a TPM quote.

The ***Cloud Agent*** is the component that runs on the cloud node and provides information about its current integrity state by sending IRs.

The ***Tenant*** represents the customer (human or organization) of the IaaS resources. It kicks off the Keylime framework, providing to the agent an encrypted payload that contains information to start up his service and to the CV all the information to attest the integrity state of the cloud node that runs the service.

The ***Software CA*** is a software-only certification authority whose purpose is to link trust and integrity measurements rooted in the TPM with the higher-level security services, avoiding the need to make each service trusted computing-aware.

The ***Revocation Service*** completes the linkage between the trusted computing services and higher-level security services. When a *CV* detects an untrusted cloud node, the *revocation notifier* sends a "revocation event" to the *software CA* and to all the cloud nodes registered for this service.

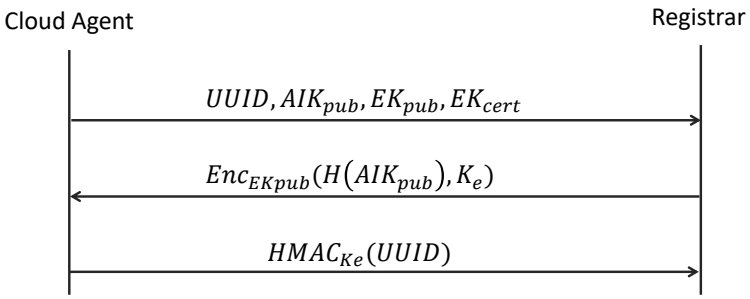

Figure 5.3. Physical node registration protocol (source: [37]).

Upon receiving this event, the *software CA*, that hosts a CRL service, revokes the software identity key corresponding to the untrusted cloud node by publishing an updated CRL, while the cloud nodes execute specific scripts, allowing higher-level security services to automatically react to this change.

One of the purposes of Keylime is to protect tenant's sensitive data from the cloud provider. These data may be software identity keys or configuration metadata like a `cloud-init` script. For this reason, at the beginning of the process the *tenant* generates a fresh symmetric key K_b and uses it to encrypt, with AES-GCM algorithm, the sensitive data d to be passed to the node, denoted $Enc_{K_b}(d)$. Then, the *tenant* requests the IaaS provider to instantiate a new resource (a physical or virtual node) and sends $Enc_{K_b}(d)$ as part of the resource creation. Upon creation, the provider returns the UUID for the new node and the IP address to which the node can be reached.

The Keylime framework can be subdivided in the following operational phases:

1. the *Physical Node Registration Protocol*;
2. the *Three Party Bootstrap Key Derivation Protocol*;
3. the *Continuous Remote Attestation*;
4. the *Revocation Framework*.

Physical Node Registration Protocol

The initial interactions of the Keylime framework concern the *Registration Protocol*, implemented by leveraging the existing TCG standard for the creation and validation of AIK keys. When the cloud agent starts up, it contacts the registrar for performing the enrollment of the standard credentials of the TPM installed in the system. As represented in figure 5.3, the cloud agent sends to the registrar its UUID, along with the AIK_{pub}, the EK_{pub} and the EK_{cert} of the TPM. The registrar stores these information and challenges the cloud node to prove that it owns the EK_{priv} and the AIK_{priv} corresponding to the public counterpart that it received. The registrar creates the challenge in this way: it generates an ephemeral symmetric key K_e, it computes a hash of AIK_{pub} (denoted $H(AIK_{pub})$) and it encrypts these two information with EK_{pub}. When the cloud agent receives the registrar's challenge, it passes this encrypted blob to the `ActivateIdentity` TPM command. The TPM will correctly decipher K_e only if it owns EK_{priv} corresponding to EK_{pub} and AIK_{priv} corresponding to AIK_{pub}. The cloud agent proves that it retrieved K_e by sending to the registrar the HMAC of its UUID computed with K_e. Upon receiving the response, the registrar recomputes the $HMAC_{K_e}(UUID)$ and, if the result is equal to the agent's response, it marks the cloud agent UUID as `active` and starts sending the cloud node's TPM credentials when asked.

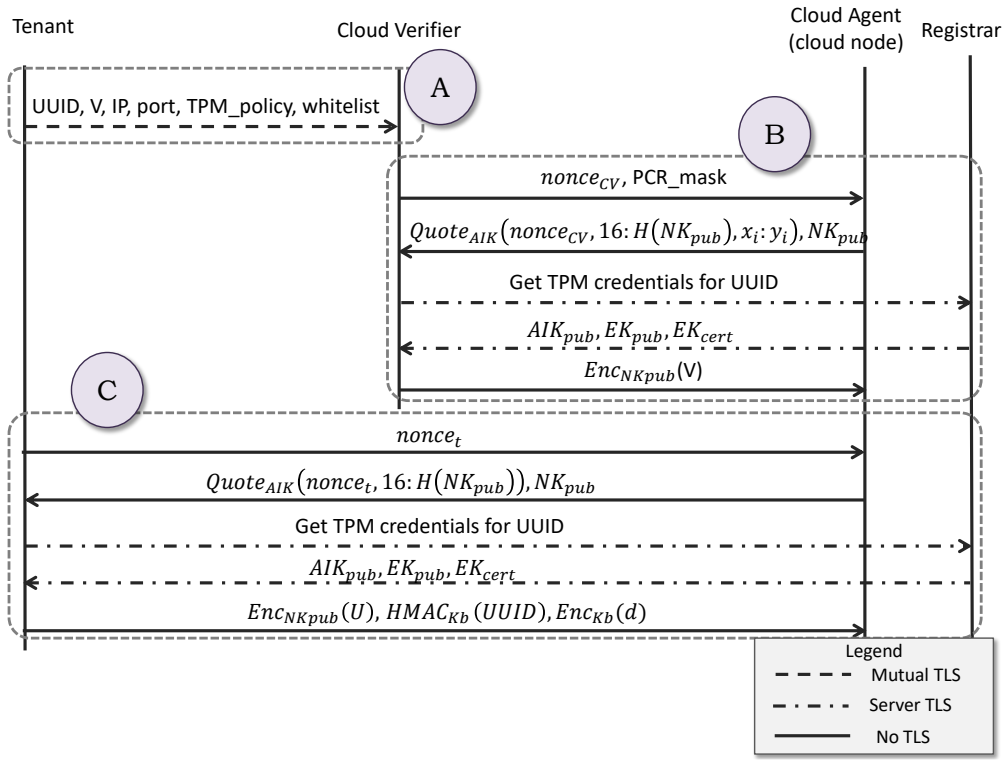

Figure 5.4. Three Party Bootstrap Key Derivation Protocol (source: [37]).

Three Party Bootstrap Key Derivation Protocol

After the *Node Registration Protocol*, the Keylime framework can execute the *Three Party Bootstrap Key Derivation Protocol*. Purpose of this protocol is to securely deliver the bootstrap key K_b to the cloud node, after having verified that it is in a trusted state. This protocol realizes two objectives:

1. demonstrating the *tenant's intent* to derive the bootstrap key K_b;
2. demonstrating the *cloud node* that will receive K_b is in a *trusted state*.

These objectives are realized through secret sharing between the tenant and the CV, in particular the tenant splits K_b into two parts U and V. These shares are obtained by generating a secure random value V of the same length as K_b and calculating $U = K_b \oplus V$. The tenant will send U directly to the cloud node to demonstrate its intent to derive K_b, and will share V with the CV, which in turn will send it to the cloud node only after having verified that the cloud node is trusted.

The interactions between tenant, CV and cloud agent can be subdivided into three phases, represented with letters A, B and C in figure 5.4. During phase A, the tenant notifies the CV that a new cloud node exists and, connecting over a secure channel, sends it the cloud agent's *UUID*, the V share, the *IP* address and *port* to which the cloud agent is reachable, the *TPM policy*, the *whitelist* and, optionally, the *MB refstate*. In particular, the *TPM policy* specifies both the PCRs that the TPM quotes have to contain and the expected values associated to those PCRs; the *whitelist* is used for validating the IMA ML and contains the list of trusted digests for the configuration files and the programs running on the cloud node; the *MB refstate* is the "Measured Boot reference state" and is used for validating the Measured Boot ML. After completing phase A, the tenant and the CV begin the attestation protocol in parallel (represented by phases B and C respectively), at the end of which the cloud agent receives U from the tenant and V from the CV. Since the cloud agent does not have a certified software identity key to establish secure

communications, it generates an ephemeral asymmetric key NK and sends the public part of this key, NK_{pub}, to the CV and the tenant, so that they can use it to cipher V and U rispectively and transmit them securely over an untrusted network. In order to prove the authenticity of NK_{pub} to the tenant and to the CV, the cloud agent extends NK_{pub} in a freshly reset PCR 16 and adds the PCR 16 in the TPM quotes. In this way, the identity of NK_{pub} is bound to the TPM identity and this allows the tenant and CV to authenticate NK by validating the TPM quote.

Now the interactions of phase B wil be examined in detail. The CV sends a request for a TPM quote to the cloud agent, specifying a fresh nonce ($nonce_{CV}$) and a mask (PCR_mask) that indicates the PCRs that the TPM quote has to contain. The node sends back NK_{pub} along with the quote $Quote_{AIK}(nonce_{CV}, 16 : H(NK_{pub}), x_i : y_i)$, where $16 : H(NK_{pub})$ represents the PCR 16 containing the hash of NK_{pub}, while $x_i : y_i$ represent the PCRs requested by the CV with their respective values. Then, the CV asks the registrar for the node's TPM credentials (AIK_{pub}, EK_{pub} and EK_{cert}) over a server authenticated TLS and uses AIK_{pub} to verify the authenticity of the quote; if the quote is authentic, the CV verifies that the cloud node has a trusted state by comparing the PCRs values contained in the quote with the trusted values specified in the TPM policy and by validating the IMA ML with the whitelist (the cloud agent adds in the IR the IMA ML only if PCR_mask contains the IMA PCR, typically PCR 10). The CV also verifies that the received NK_{pub} is correct by computing the hash over NK_{pub} and checking that the result is equal to the content of PCR 16. If all the verification steps are passed, the CV sends back to the cloud agent the V share, encrypted with NK_{pub}, then it starts the "Continuous Remote Attestation" phase described in the following section. Otherwise, the CV does not send V and sets the state associated to the cloud node as `INVALID_QUOTE`; even if the tenant already sent the U share (phase C), there is no concern of leaking U to an untrusted node because K_b is different for each cloud node.

The interactions of phase C occur in parallel to those of phase B and, except for some small differences, are similar. The tenant requests a TPM quote to the cloud agent, specifying a fresh nonce ($nonce_t$) and an empty PCR_mask. The empty PCR_mask is due to the fact that, differently from the CV, the tenant does not use the quote to verify the trusted state of the node but only to verify the identity of the TPM in order to authenticate NK_{pub}. The cloud agent sends back NK_{pub} along with $Quote_{AIK}(nonce_t, 16 : H(NK_{pub}))$, a TPM quote containing only PCR 16 for validating NK_{pub}. The tenant asks the registrar to provide the TPM credentials related to the node $UUID$, then it uses AIK_{pub} to verify the authenticity of the quote, EK_{pub} and EK_{cert} to verify that they correspond to an authentic TPM. In particular, the tenant verifies that:

1. the public key contained in EK_{cert} is equal to EK_{pub};
2. the issuer of EK_{cert} is a trusted TPM manufacturer whose certificate is contained in a tenant's local repository (called "tpm_cert_store");
3. the signature of EK_{cert} is autentic, verifying it with the public key contained in the TPM manufacturer's certificate.

If one of the previous checks is not passed, the tenant notifies this to the CV, which sets the state of the cloud node to `TENANT_FAILED` and stops the periodic attestation on the cloud node. Moreover, the tenant will not send the U share to the cloud agent which will not be able to derive K_b and consequently to disclose the encrypted payload. Instead, if the TPM of the cloud node is authentic, the tenant verifies the validity of the quote with the AIK_{pub} key, then it verifies the correctness of the received NK_{pub} in the same way as the CV does. If NK_{pub} results authentic, the tenant sends to the cloud agent the U share encrypted with NK_{pub}, the HMAC over the node's $UUID$ computed with K_b ($HMAC_{K_b}(UUID)$) and the encrypted payload $Enc_{K_b}(d)$, if it is not been sent contextually to the IaaS resource request.

When the cloud agent receives U from the tenant or V from the CV, it verifies whether it has received both shares of K_b and, in this case, it computes $K_b = U \oplus V$. Then it checks if the derived K_b is the correct one by computing $HMAC_{K_b}(UUID)$ and verifying that it is equal to the value sent by the tenant; if so, the cloud agent uses K_b to decipher the encrypted payload and proceeds with the startup of the tenant's service. After decrypting the payload, the cloud agent deletes K_b and V while it stores U in the TPM NVRAM in order to automatically support

node's reboot or migration without the need for the tenant to interact again with the cloud node. Every time the cloud agent needs of the V share after reboot or migration, it sends CV a new NK_{pub} along with the TPM quote; upon receiving a new NK_{pub}, the CV provides the V share to the cloud agent, so that the it can recombine K_b and decipher the encrypted payload again. Obviously, the automatic reboot or migration only work if the provider has correctly and securely migrated the TPM NVRAM containing U.

Encrypted payload contents

The payload that the tenant provides to the cloud agent, ciphered with K_b, is typically an archive containing several files, regarding both the deployment of the application and the Keylime framework. The archive may contain the following files:

- *software identity keys* and *certificates* for higher-level security services;
- a bash script automatically executed after the payload is decrypted; by default, this script has to be named `autorun.sh`, but the name can be changed through the Keylime's configuration file; this script acts as a "deploy-hook", thus it has to contain instructions to deploy the application, for example a call to `ansible-playbook`;
- the *revocation certificate*, used to check the signature over the *revocation events* received from the *Revocation Notifier*; a *revocation event* notifies that one of the cloud nodes in the tenant's cluster became untrusted and it is accepted only if it has a valid signature;
- a list of Python scripts, whose name begin with `local_action_<any_name>.py`, executed when the cloud agent receives a *revocation event* with a valid signature; these scripts should contain instructions for notifying higher-level security services that something changed and for fencing off the compromised machine; for example, the scripts could contain calls with Kubectl, make some changes to IPtables, close VPN tunnels and so on;
- a text file called `action_list`, containing the names, separated by commas, of all the scripts `local_action_<any_name>.py` contained in the payload; this file specifies the order in which the "local_action" scripts will be executed and, if a script is contained in the payload but it is not inserted in this list, it will be ignored.

Continuous Remote Attestation

Once the *Three Party Bootstrap Key Derivation Protocol* terminates, the Keylime framework moves into the third phase, which is *Continuous Remote Attestation*. After having verified that the remote system performed a trusted boot, so that the application will be deployed in a trusted environment, the CV periodically polls the cloud node to monitor its integrity state and verify that it remains trusted over time, by relying on the integrity measurements that IMA performs on the applications launched in the system. As shown in figure 5.5, the CV periodically requests a new IR to the cloud agent and performs the IR validation, so it is able to detect any integrity change that happens in the system. In particular, in order to validate the IR, the CV verifies that:

- the quote signature is valid, checking it with the AIK_{pub} key provided by the registrar;
- the quote contains all PCRs specified in the TPM policy;
- the quote contains PCR 16 (not specified in the TPM policy) and its content is equal to the digest (computed with the hash algorithm associated to the PCR 16) of the NK_{pub} sent by the cloud agent during the *Bootstrap Key Derivation Protocol* and stored in the local DB;
- the IMA ML matches the PCR 10 value;
- the measurement events contained in the IMA ML match the whitelist;

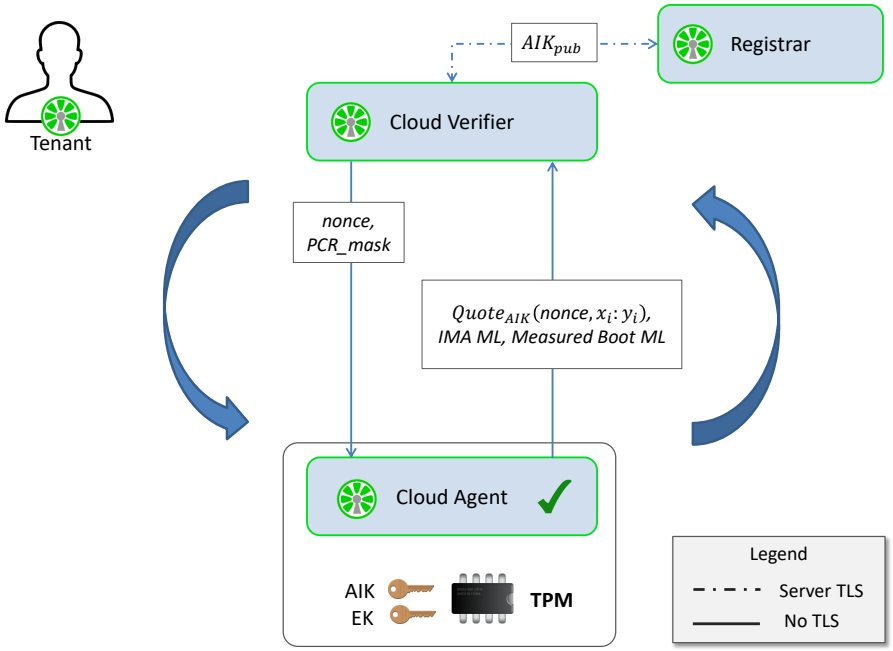

Figure 5.5. Continuous Remote Attestation.

- the PCR values specified in the Measured Boot (MB) ML match the corresponding PCR values contained in the quote;
- the MB ML matches the *MB refstate* provided by the tenant.

The time interval that elapses between an attestation request and the next one determines the latency with which the detection of compromise occurs; by default this time interval is of two seconds, but it can be set to a different value via the Keylime's configuration file. However, this latency will be at least 500 ms, considering that the TPM_quote operation alone could take more than 500 ms.

Revocation Framework

As soon as the CV detects that a cloud node is untrusted, it triggers the *Revocation Framework*, whose interactions are represented in figure 5.6. This framework relies on the *revocation notifier*, a ZeroMQ server spawned by the CV at start up. This server implements the *publish/subscribe* pattern, in particular the CV publishes a new *revocation event* by sending a signed message to the ZeroMQ server, which in turn forwards it to all its subscribers. For doing this, ZeroMQ server binds two sockets:

- a unix socket for Inter Process Communication with the CV;
- a TCP/IP socket, to which all the subscribers that want to receive *revocation events* have to connect and await to receive notifications.

The subscribers can be:

- the software CA which, upon receiving a revocation message for an untrusted cloud node, revokes the certificate of the identity key owned by that node and publishes an updated CRL;
- the cloud agents which, upon receiving a new revocation message, execute the "local_action" scripts for ring-fencing the untrusted node;
- other tools that want to be notified about the events related to the trusted computing layer.

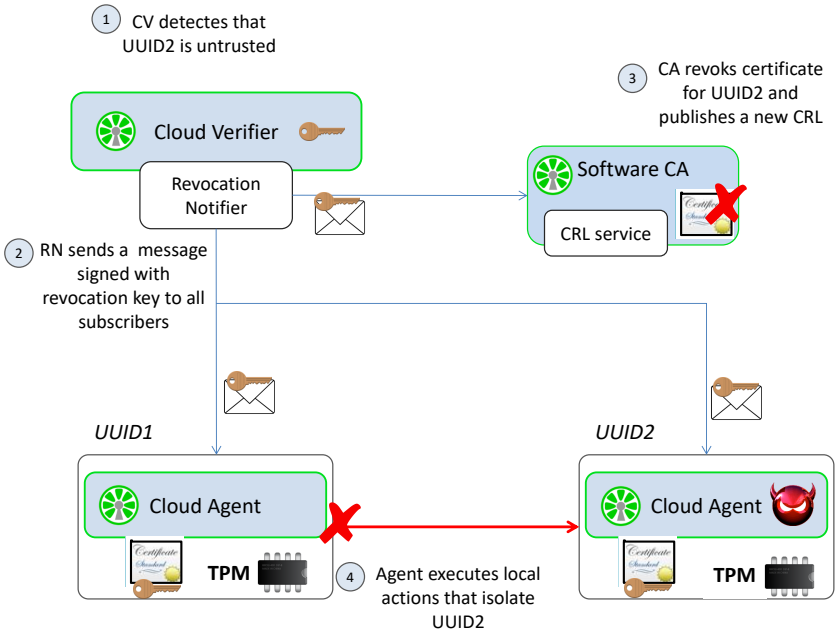

Figure 5.6. Revocation Framework.

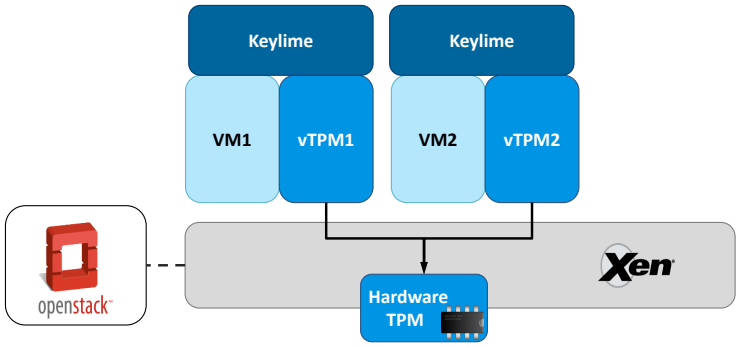

Figure 5.7. vTPM supports in Xen hypervisor (source: [41]).

5.2.3 Layered Architecture for Virtual Nodes

Keylime designers expanded the architecture previously exposed to work also across the layers of virtualization, considering the widespread use of virtualization in today's IaaS infrastrutcures. In this scenario, tenant nodes are VMs managed by VM orchestrators like Amazon EC2 or OpenStack. Each VM needs a root of trust on which to build trusted computing services, but physical TPMs, having limited performance and resources, cannot be used to directly serve the VMs because a direct multiplexed use of them would not scale to the numbers of VMs typically hosted on modern systems. Keylime tries to overcome this problem by relying on vTPMs, described by Berger et al. [42] and implemented in the Xen hypervisor. In this implementation, each VM has its own vTPM running in a separate Xen domain, isolated from all other VMs running on the hypervisor, as shown in figure 5.7. The vTPM interface is the same as a physical TPM, except for the addition of the *deep-quote* operation, that binds the vTPM quote to the hardware TPM quote. A deep-quote consists of a vTPM quote and a hardware TPM quote, where the first is computed by using the *nonce* provided by the verifier and is signed with a vAIK generated by the vTPM, while the second is computed by using as *nonce* the hash of the vTPM quote and is signed with an AIK generated by the physical TPM. To assure a chain of trust rooted in the

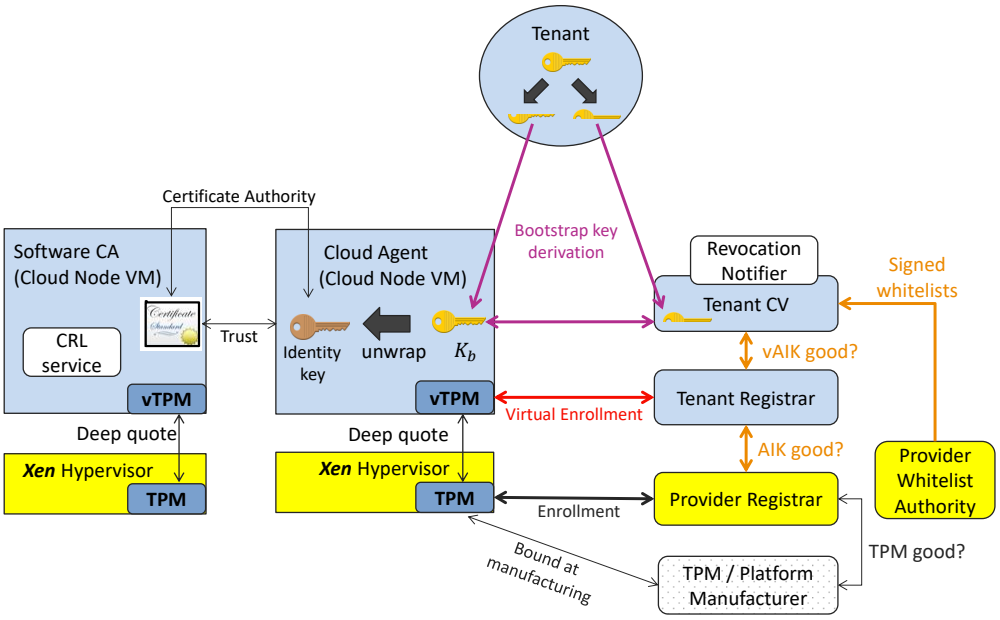

Figure 5.8. Keylime layered architecture (source: [37]).

hardware TPM, the IaaS provider has to add some additional components in its infrastructure and allow the tenant's trusted computing services to query them, in particular:

- a ***Provider Registrar***, whose purpose is to provide the credentials of the physical TPM installed in the system that hosts the virtual cloud node;
- a ***Provider Whitelist Authority Service***, that publishes an up-to-date signed list of the integrity measurements of the provider infrastructure;
- optionally, a ***Provider CV*** that attests the provider infrastructure.

The layered Keylime architecture, represented in figure 5.8, conceptually works the same way as the simplified architecture, with little modifications to some protocols to support the presence of the vTPMs.

Virtual Node Registration Protocol

The *Virtual Node Registration Protocol*, represented in figure 5.9, involves the cloud agent running in a VM, the tenant registrar and the provider registrar. It begins when the cloud agent sends the node's *UUID*, the $vAIK_{pub}$ and the vEK_{pub} of the vTPM to the tenant registrar (the vEK_{cert} is empty because the TPM is virtual and has no manufacturer). The tenant registrar generates an ephimeral key K_e and responds with $Enc_{vEK_{pub}}(H(vAIK_{pub}), K_e)$. The cloud agent decrypts K_e by means of the `ActivateIdentity` command of its vTPM, extends the node's *UUID* and the vTPM credentials ($vAIK_{pub}$ and vEK_{pub}) in a freshly reset vPCR 16, computes $SHA1(K_e)$ and uses it as nonce of a deep-quote that contains the vPCR 16. With this deep-quote, the cloud agent demonstrates to the tenant registrar that:

1. the vTPM correctly disclosed K_e, so it owns vEK_{priv} and $vAIK_{priv}$;
2. there is a link between the vTPM credentials (vEK, vAIK) and a physical TPM in the provider's infrastructure.

Upon receiving the deep quote, the tenant registrar asks the provider registrar to provide the credentials of the physical TPM installed on the system where the cloud node *UUID* is running,

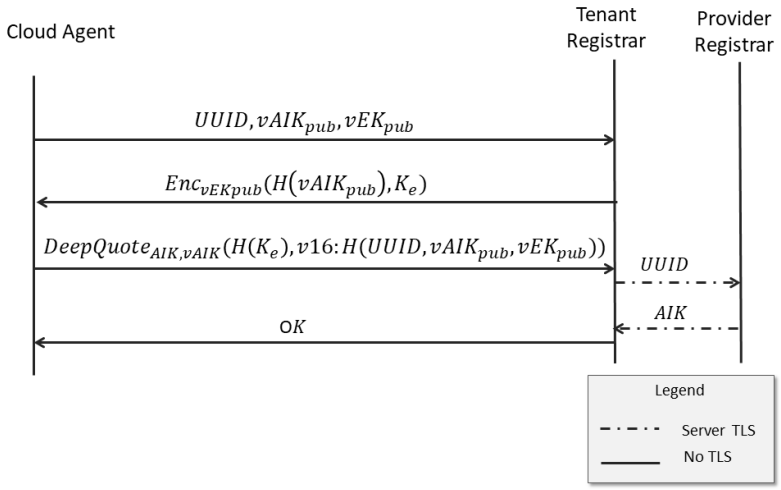

Figure 5.9. Virtual node registration protocol (source: [37]).

then uses AIK_{pub} and $vAIK_{pub}$ to check the validity of the deep-quote signatures. It checks also that the deep-quote's nonce is $SHA1(K_e)$ and that the content of vPCR 16 matches the information previously provided by the cloud agent. If the deep-quote passes the verifications, the tenant registrar marks the cloud node as *active* and starts to provide the virtual and physical TPM credentials when asked.

Virtual Node Remote Attestation

As described for the simplified architecture, the CV attests the cloud node at the beginning, in order to check the trusted boot and complete the *Three Party Bootstrap Key Derivation Protocol*; then it continues to monitor the cloud node to detect any integrity compromise that can occur during the runtime. Since the cloud node runs in a layered environment, the CV has to attest both the cloud node's VM and the hosting system; so, when it asks a deep-quote to the cloud agent, it sends in the request:

- a fresh *nonce*;
- the PCR_mask that specifies PCRs of the physical TPM contained in the deep-quote;
- the vPCR_mask (derived from the vTPM policy sent by the tenant) that specifies the vPCRs of the vTPM contained in the deep-quote.

Then, in order to verify the deep-quote, the CV needs of both the $vAIK_{pub}$ and the AIK_{pub}; it retrieves them from the tenant registrar, which obtained both physical and virtual TPM credentials during the *Virtual Node Registration Protocol*.

Chapter 6

Docker containers attestation with Keylime

This chapter exposes the proposed solution for performing remote attestation of Docker containers. The integrity attestation of containers deployed on a host system is currently achievable through various solutions already presented to the scientific community, as explained in the chapter 4. However, these solutions are not always satisfactorily applicable and their implementations are based on TPM 1.2, a currently deprecated specification. The solution proposed in this thesis aims to improve the performance of the previous solutions and overcome some of their limitations. The main objective of the thesis was to be able to individually attest Docker containers running on a node, relying on TPM 2.0. The integrity attestation of individual containers is essential in order to isolate and stop any untrusted container in the platform and let other trusted containers continue to provide their functionalities without being interrupted. This was achieved by creating specific IMA templates, described in section 6.3.1 and 6.3.2, that is, by inserting in the ML entries appropriate fields that can unequivocally determine their association to a particular container. The prototype of the proposed solution is based on the Keylime, a RA framework whose architecture was exposed in chapter 5 and which is based on TPM 2.0 as hardware RoT. Various changes have been made to the Keylime code in order to add support to the new templates created, improve the latency times of the RA process and allow the use of hash algorithms other than SHA-1 for the integrity verification of the IMA ML. The overall architecture for performing remote attestation of Docker containers is shown in section 6.2, while the details on the changes introduced in the Keylime modules are described in section 6.4.

6.1 Approach

As described in chapter 4, recently researchers proposed several solutions that address the problem of measuring the integrity status of services deployed in a lightweight virtualization environment. All of them are based on the TPM for building trust in the platform, on the Trusted Boot for measuring the boot of the platform and on IMA for measuring the files accessed at runtime. As already discussed, a RA mechanism that aims to monitor the integrity status of container-based environments should have the following desirable features: measuring the integrity status of a container and the underlying host, generating a tamper-proof integrity evidence, ensuring efficiency and scalability, guaranteeing the privacy among containers in the case of multi-tenant attesting systems. The solutions proposed up to now do not always satisfy all these objectives and their implementations are based on the OAT framework which only supports TPM 1.2, currently substituted by the TPM 2.0 specification.

Focus of this work is to create a solution that allows to attest Docker containers deployed in a single-tenant host system, trying to satisfy the characteristics listed above and using TPM 2.0 as hardware RoT. In order to realize this objective, two approaches were examined. The first

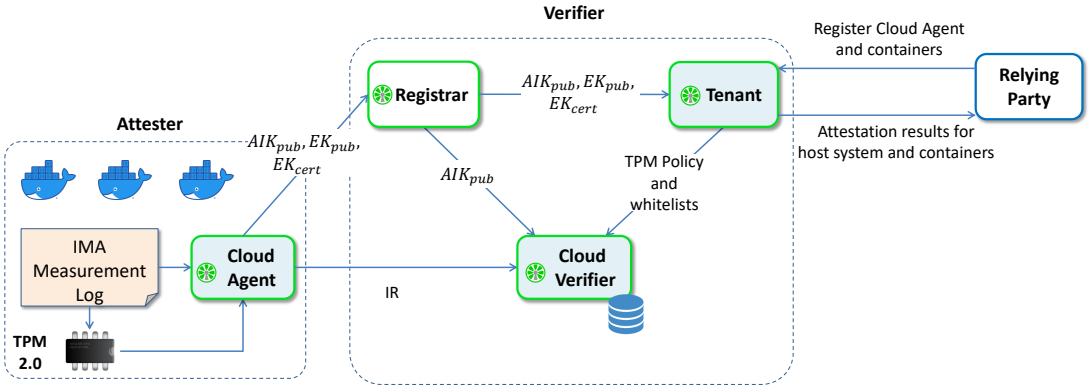

Figure 6.1. Architecture of the proposed solution. The Keylime components with a light blue background are those affected by code modifications.

approach, chosen by DIVE and Container-IMA solutions (described in sections 4.2 and 4.3 respectively), starts from considering containers simply as processes running on the host system and relies on the integrity measurements taken by the IMA module of the host kernel, as described in chapter 3. In this approach it happens that MEs generated by processes belonging to the different containers and to the host system are stored in the same ML; this implies that, in order to attest a specific container, we have to be able to state which MEs belong to that container and which one to the host system. The second approach considers containers as VMs and builds for them separate RoTs via vTPMs [43]. Since a vTPM aggregates the integrity measurements of a VM inside a vPCR located in the user-space memory, the measurements do not have a hardware-based RoT and the solutions proposed so far to anchor a vTPM to a hardware TPM do not guarantee good performance. Moreover, this approach is not transparent from a container perspective because it requires to modify the container image for including a vTPM driver. Analyzing these two possible approaches, we chose to follow the first one because it better meets the TCG specification, which assume a one-to-one relationship between the operating system and the TPM chip [44].

6.2 Architecture

The overall architecture to perform remote attestation with the proposed solution is based on Keylime, whose architecture is described in section 5.2.2; the changes introduced in the framework did not affect its architecture but only some parts of the implementation of Cloud Agent, Cloud Verifier and Tenant, as shown in figure 6.1. The components of the architecture are:

- the *Attester* is the target of the RA process and is the platform which hosts containers belonging to a single tenant; it supports trusted boot to ensure the integrity of all the boot components, relies on the IMA module to measure the integrity of services running on the platform, both containerized and non-containerized ones, and is equipped with a TPM 2.0 chip to provide hardware RoT to measurements;
- the *Cloud Agent* is a service running on the Attester; it receives attestation requests, to which it responds with an IR containing a TPM quote and the IMA ML;
- the *Cloud Verifier* is the component in charge of deciding the trustworthiness of the host system and the containers running on it;
- the *Registrar* receives the TPM's credentials from the Cloud Agent and provides them to the CV and the Tenant, in order to allow them to check the authenticity of the IR;
- the *Tenant* registers a new Cloud Agent in the CV and sends to it the TPM policy, specifying the TPM's PCRs to be checked, and the whitelists, containing the expected values of the measurement events generated by physical host and containers;

```
$ cat /proc/3170/cgroup
12:memory:/docker/5cbc6f87377485c0355317cedf536979cde8c38023fdff42e2cf2093ec20ea0b
11:devices:/docker/5cbc6f87377485c0355317cedf536979cde8c38023fdff42e2cf2093ec20ea0b
10:cpuset:/docker/5cbc6f87377485c0355317cedf536979cde8c38023fdff42e2cf2093ec20ea0b
9:perf_event:/docker/5cbc6f87377485c0355317cedf536979cde8c38023fdff42e2cf2093ec20ea0b
8:cpu,cpuacct:/docker/5cbc6f87377485c0355317cedf536979cde8c38023fdff42e2cf2093ec20ea0b
7:hugetlb:/docker/5cbc6f87377485c0355317cedf536979cde8c38023fdff42e2cf2093ec20ea0b
6:blkio:/docker/5cbc6f87377485c0355317cedf536979cde8c38023fdff42e2cf2093ec20ea0b
5:pids:/docker/5cbc6f87377485c0355317cedf536979cde8c38023fdff42e2cf2093ec20ea0b
4:freezer:/docker/5cbc6f87377485c0355317cedf536979cde8c38023fdff42e2cf2093ec20ea0b
3:net_cls,net_prio:/docker/5cbc6f87377485c0355317cedf536979cde8c38023fdff42e2cf2093ec20ea0b
2:rdma:/
1:name=systemd:/docker/5cbc6f87377485c0355317cedf536979cde8c38023fdff42e2cf2093ec20ea0b
0::/system.slice/containerd.service
```

Figure 6.2. Content of `cgroup` file related to the init_process_pid 3170 of a container with ID `5cbc6f873774`. and result of the command `systemd-cgls name=systemd`.

- the *Relying Party* represents the tool or human user that manages the containerized services running on the Attester and wants to check their integrity state; it registers the remote services in the Keylime framework through the Tenant module, from which it will later receive the attestation results.

6.3 IMA patches

The first problem faced was how to allow a verifier to recognize if a given ME belongs to a container or to the host system. IMA, in fact, stores all MEs in the same ML and the builtin IMA templates (`ima-ng`, `ima-sig` and so on) do not contain fields which allow a verifier to determine if a given ME belongs to a container or to the host system and, in the first case, to which specific container it is associated. DIVE solved this problem by defining an IMA template containing the `dev-id` field, which allows to determine if the accessed file belongs to a particular container or to the host system. However, this solution has some drawbacks. It works only if Docker is configured to use Device Mapper as storage driver; moreover, if a container is terminated and a new one is created, the new container acquires the same `dev-id` as the previous one, so all the MEs generated by the terminated container are inherited by the new one, making it impossible to correctly assess the integrity state of the new container. Container-IMA, instead, uses the `mount namespace` to identify the MEs generated by a container; however, this identifier has the same problem as `dev-id` if used alone inside an IMA template, because `mount namespace` numbers assigned to terminated containers are immediately "recycled" by the Linux kernel when new containers are created.

In order to overcome this issue, the first step was the deepening of what characterizes a container within the Linux kernel, in particular the resource isolation features on which the container technology is based, such as *cgroups*, *namespaces* and *kernel capabilities* [33]. Working with an Ubuntu 20.04 operating system on which Docker version 20.10.5 was installed, it was observed that Docker creates, for each new container, control groups with name equal to the *container full-ID* which is an identifier of 256 random bits, used by Docker to identify a container within its data structures. What is called *container ID* corresponds the most significant 48 bits of the container full-ID, that is, the most significant 12 hexadecimal digits. As discussed in paragraph 4.1, Docker is a containerization technology that creates *process containers*, that is each Docker container targets a main application, whose process can be called the *init_process* of the container. All processes running inside a container are children of the init_process and the control groups assigned to the init_process are inherited by all its children processes. Moreover, a process running in a container neither can change its namespaces or cgroups nor can launch children processes in other namespaces or cgroups; thus, all processes running inside a given container have the same cgroups. The figure 6.2 shows all the cgroups assigned to the init_process of a container with

```
$ systemd-cgls name=systemd
Controller name=systemd; Control group /:
├─ 1645 bpfilter_umh
├─ docker
│  ├─ 987d703b44240bf5193fe6e6408f07e944317fac733d5d75bc819427a0dd5fd2
│  │  └─ 2990 /bin/bash
│  └─ 5cbc6f87377485c0355317cedf536979cde8c38023fdff42e2cf2093ec20ea0b
│     └─ 3170 /bin/bash
├─ user.slice
│  └─ user-1000.slice
│     ├─ user@1000.service
│     │  ├─ gsd-xsettings.service
│     │  │  └─ 2165  /usr/libexec/gsd-xsettings
│     │  ├─ gvfs-goa-volume-monitor.service
│     │  │  └─ 1863  /usr/libexec/gvfs-goa-volume-monitor
  ...
```

Figure 6.3. Snippet of the `name=systemd` cgroup hierarchy on a system that runs two Docker containers.

ID `5cbc6f873774`, stored in the file `/proc/3170/cgroup`, where `3170` is the process PID. The file contains a row for each cgroup hierarchy of which the process is member; each row has three colon-separated fields [33]:

$$hierarchy - ID \ : \ controller - list \ : \ cgroup - path$$

where:

- *hierarchy-ID* is the unique hierarchy ID number for cgroups hierarchies of version 1; it's the value 0 for the cgroups hierarchy of version 2;
- *controller-list* contains a comma-separated list of controllers bound to the hierarchy for cgroups hierarchies of version 1; it's the empty string for the cgroups hierarchy of version 2;
- *cgroup-path* contains the pathname of the cgroup in the hierarchy, relative to the mount point of the hierarchy.

As we can see, all cgroups version 1, except `rdma`, have the container full-ID in the cgroup-path. Instead, figure 6.3 represents a snippet of the cgroup hierarchy related to the `name=systemd` controller, on a system where two Docker containers, with ID `5cbc6f873774` and `987d703b4424`, are running.

6.3.1 IMA template `ima-cgn`

On the basis of the observations exposed above, it was chosen to use the cgroups of the process that generated the ME for being able to associate its corresponding ML entry to a particular container or to the host system. So we defined a new IMA template, called `ima-cgn`, which adds a field containing the name of a cgroup to the fields present in the `ima-ng` template. The format string defined for `ima-cgn` is `"cgn|d-ng|n-ng"`, where `cgn` is the cgroup name, `d-ng` is the digest of the file data and `n-ng` is the file path. As shown in figure 6.2, Docker creates cgroups with name equal to the container full-ID for several control subsystems; among the various subsystems, in the implementation it was chosen to use `name=systemd` cgroup, with hierarchy ID = 1. The details regarding the implementation of `ima-cgn` template are described in section C.1.1.

The figure 6.4 shows some ML entries formatted with `ima-cgn` template; each entry is made up of fields PCR index, SHA-1 template-hash and template name, followed by the specific fields of `ima-cgn` defined in the format string, which are cgroup-name, filedata hash and file path. The figure highlights the entries belonging to the container with ID `5cbc6f873774`, which are distinguished, by means of the `cgroup-name` field, from those generated by the container `6fae267e6042` and by the host system.

PCR	template-hash	templ-name	cgroup-name	filedata-hash	filename-hint
10	9cab3[...]ab7e3	ima-cgn	/	sha256:c20ff[...]8dd	boot_aggregate
10	1590d[...]94e7	ima-cgn	/	sha256:b0537[...]fca	/usr/bin/kmod
10	767f0[...]8802	ima-cgn	dev-hugepages.mount	sha256:f2909[...]4bf	/usr/bin/mount
..	...	...	...	...	...
10	92edf[...]ffd51	ima-cgn	5cbc6f873774[...]a0b	sha256:04a48[...]ea9	/usr/bin/bash
10	4f9f2[...]ec917	ima-cgn	5cbc6f873774[...]a0b	sha256:69ba8[...]bd8	/usr/lib/x86_64-linux-gnu/ld-2.31.so
10	07ab4[...]5596c	ima-cgn	5cbc6f873774[...]a0b	sha256:42537[...]6fa	/usr/lib/x86_64-linux-gnu/libtinfo.so.6.2
...	...	...	...	...	...
10	29d8a[...]0b2	ima-cgn	6fae267e6042[...]5e1	sha256:175a5[...]54c	/usr/bin/groups
10	6d5c7[...]a511a	ima-cgn	6fae267e6042[...]5e1	sha256:e40a[...]139	/usr/bin/dircolors
10	b18a1[...]936	ima-cgn	user.slice	sha256:35f94[...]9a3	/usr/../systemd-cgl
10	38125[...]629	ima-cgn	user.slice	sha256:37591[...]e34	/usr/bin/less
...	...	...	...	...	...

Figure 6.4. Example of IMA ML with `ima-cgn` template, generated on a system after creating two Docker containers with ID `5cbc6f873774` and `6fae267e6042` via `docker run -it ubuntu /bin/bash`.

PCR	templ-hash	template-name	dependencies	cgroup-name	filedata-hash	filename-hint
10	ea99f9d4[...]	ima-dep-cgn	swapper/0:swapper/0	/	sha256:7fd05f333e1b [...]	boot_aggregate
...	...	...		...	...	...
10	8af8cfc4f[...]	ima-dep-cgn	runc:/usr/bin/containerd-shim-runc-v2:/usr/lib/systemd/systemd:swapper/0	8b2ad985209b510bfd466aea87c11[...]	sha256:04a484f27a4b [...]	/usr/bin/bash
10	01c73d7f[...]	ima-dep-cgn	/usr/bin/bash:/usr/bin/containerd-shim-runc-v2:/usr/lib/systemd/systemd:swapper/0	8b2ad985209b510bfd466aea87c11 [...]	sha256:69ba80c71bff [...]	/usr/lib/x86_64-linux-gnu/ld-2.31.so
10	07bb4c6 [...]	ima-dep-cgn	/usr/bin/bash:/usr/bin/containerd-shim-runc-v2:/usr/lib/systemd/systemd:swapper/0	8b2ad985209b510bfd466aea87c11 [...]	sha256:425378a0c71b[...]	/usr/lib/x86_64-linux-gnu/lbtinfo.so.6.2
...	...	...	...	...	...	...

Figure 6.5. Example of IMA ML with `ima-dep-cgn` template, generated on a system after creating a Docker container with ID `8b2ad985209b` via `docker run -it ubuntu /bin/bash`.

6.3.2 IMA template `ima-dep-cgn`

The `cgroup-name` field allows to identify the entries associated to containers in the ML. However, it was decided to strengthen the link of an entry to a container through an additional field that shows the dependencies of the process that generated the ME. So we created another template, `ima-dep-cgn`, which adds the `dep` field, containing the paths of all the ancestors of the process that generated the ME, to the fields already defined by `ima-cgn`. In this way a verifier can consider an entry as belonging to a container only if it has both the `cgn` field with the format of a container full-ID and the `dep` field containing the container runtime service in the dependencies list; otherwise, a verifier should consider the entry as belonging to the host system.

The figure 6.5 shows an example of ML generated with the `ima-dep-cgn` template, where the highlighted entries belong to a container with ID `8b2ad985209b`. The format string of the `ima-dep-cgn` template is `"dep|cgn|d-ng|n-ng"`, which means that the entry mandatory fields

PCR	template-hash	template-name	dependencies	cgroup-name	filedata-hash	filename-hint
10	sha256:ea99f9d40d63[...]	ima-dep-cgn	swapper/0:swapper/0	/	sha256:7fd05f333e1b[...]	boot_aggregate
10	sha256:1590da5718a7[...]	ima-dep-cgn	kworker/u8:3:kthreadd:swapper/0	/	sha256:b053785f6308[...]	/usr/bin/kmod
10	sha256:8af8cfc4caa3b66d[...]	ima-dep-cgn	runc:/usr/bin/containerd-shim-runc-v2:/usr/lib/systemd/systemd:swapper/0	8b2ad985209b510bfd466aea87c11[...]	sha256:04a484f27a4b[...]	/usr/bin/bash
10	sha256:01c73d70f24cabd[...]	ima-dep-cgn	/usr/bin/bash:/usr/bin/containerd-shim-runc-v2:/usr/lib/systemd/systemd:swapper/0	8b2ad985209b510bfd466aea87c11 [...]	sha256:69ba80c71bff[...]	/usr/lib/x86_64-linux-gnu/ld-2.31.so
10	sha256:7bb4c6ce48f64c3[...]	ima-dep-cgn	/usr/bin/bash:/usr/bin/containerd-shim-runc-v2:/usr/lib/systemd/systemd:swapper/0	8b2ad985209b510bfd466aea87c11 [...]	sha256:425378a0c71b[...]	/usr/lib/x86_64-linux-gnu/libtinfo.so.6.2
...	...	...	...	...	...	...

Figure 6.6. Example of IMA ML with `ima-dep-cgn` and template-hash computed with the SHA-256 algorithm.

(PCR index, template-hash and template-name) are followed by the colon-separated list of dependencies, the cgroup name, the filedata hash and the file path. All the implementation details regarding this template are described in section C.1.2.

6.3.3 The template-hash field

Another important point of the thesis work was to make sure that the solution exploited the potential offered by TPM 2.0. Differently from TPM 1.2, a TPM 2.0 chip has a number of configurable PCR banks, to which any of the hash algorithms supported by the TPM can be associated; typically, a TPM 2.0 chip has at least one PCR bank for SHA-1 and another for SHA-256. Keylime, while supporting TPM 2.0, uses the PCR 10 from the SHA-1 bank for checking the integrity of the IMA ML, as required by TPM 1.2. Going deeper into the IMA code, it was observed that IMA supports the algorithm-agility of TPM 2.0, and therefore it is possible to use any PCR bank for the integrity verification of the ML. However, when IMA writes the template-hash field in the ML, it uses only the one computed with the SHA-1 algorithm and extended in the SHA-1 bank, without allowing the user to choose the template-hash to be recorded in the ML. While this opportunity is not strictly necessary to perform the ML integrity check, it was decided to overcome this limitation and to give the user the option to specify the template-hash to record in the ML. So a new kernel command line parameter has been introduced, `ima_template_hash=`, through which the user can specify one of the hash algorithms supported by the Linux kernel; it was also decided that, if the selected algorithm does not correspond to a PCR bank in the TPM, the value written in the ML will be a padded SHA-1.

The figure 6.6 shows an example of ML in which the template-hash field has been computed with the SHA-256 algorithm. The details about the implementation of the `ima_template_hash=` kernel boot parameter are presented in section C.1.3.

6.4 Keylime changes

As already mentioned at the beginning of this chapter, Keylime was chosen as RA framework since it, differently from OAT, supports TPM 2.0. The second part of the thesis work concerned the changes that needed to be made to the Keylime code in order to support the registration of containers inside the CV and their attestation by means of the IMA templates described in the

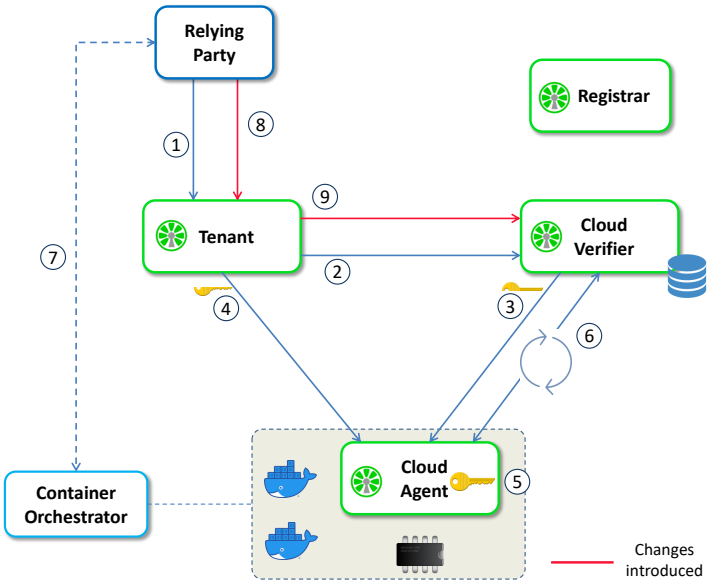

Figure 6.7. Workflow of the registration process and the periodic attestation of a host system on which Docker containers are running.

previous sections. The modifications pertained the Cloud Agent, the CV and the Tenant modules. The details about the REST APIs exposed by each Keylime module are described in appendix D.

6.4.1 Registering Docker containers

The registration of Docker containers in the CV can occur at the same time as the registration of the Cloud Agent or later. In the following description, it is assumed that the IMA module of the attesting system has been configured to use the `ima-dep-cgn` template. The figure 6.7 represents the workflow of the registration process of Docker containers launched on the attesting system after verifying that it performed a trusted boot.

(1) The Relying Party kicks off the process by requesting the registration of a new Cloud Agent to the Tenant; the body of the request specifies the UUID and the IP address of the remote node, the payload containing the `autorun.sh` script for deploying Docker containers, the whitelist, the exclude list and the TPM policy for attesting the host system. It was decided to add another parameter in the registration request, `allow_unknown_containers`, for specifying if the CV has or not to evaluate the detection, on the attesting system, of unregistered containers as an integrity violation; in the scenario described in this section, the Relying Party sends the registration request of Cloud Agent with this parameter set to `1`, in order to manage the time interval between the moment in which the containers are launched on the attesting system and the moment in which they are registered in the CV. Then, the Tenant encrypts the payload for the Cloud Agent with a random bootstrap key K_b and starts the *Three Party Bootstrap Key Derivation Protocol* described in section 5.2.2. (2) The Tenant registers the new Cloud Agent in the CV, delivering to it all the information received from the Relying Party, with the addition of the port at which the Cloud Agent service is reachable and the V share of K_b. Then, the Cloud Agent receives (3) V from the CV and (4) U, together with the encrypted payload, from the Tenant, after that all the checks exposed in 5.2.2 were carried out by the CV and the Tenant, respectively. (5) The Cloud Agent recomposes K_b, deciphers the encrypted payload and executes the `autorun.sh` scripts that deployes the Docker containers. (6) The CV, after having delivered V, continues to periodically monitor the integrity state of the attesting system; in this phase the CV checks only the host system, because the Tenant has not yet sent the information about the containers to be attested. (7) When the Relying Party retrieves the container IDs from the container orchestrator, (8) it sends the request for registration of the containers to the Tenant, specifying for each container its ID, the whitelist and exclude list needed to attest it; moreover, if the host system is assumed

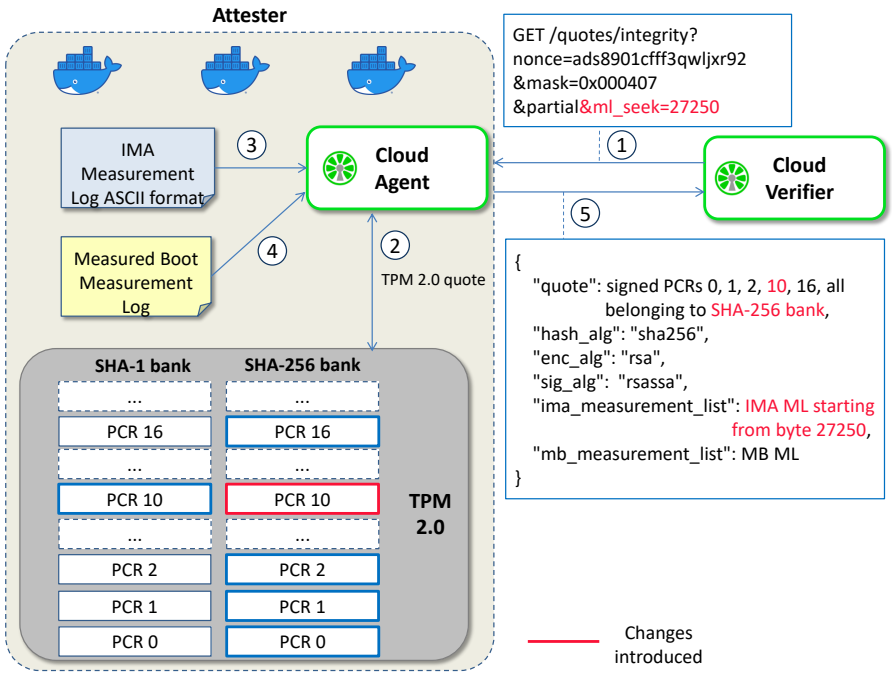

Figure 6.8. IR creation process of the proposed solution.

to execute only those containers, it specifies also `allow_unknown_containers=0` in the body of the request. (9) The Tenant forwards the registration request of the containers to the CV which, from that moment on, attests the integrity state of both the registered containers and the host system. The requests for registration of containers are done through REST APIs added to those already present in the Keylime framework and presented in appendix D.

6.4.2 Creating the Integrity Report

The figure 6.8 illustrates the interaction between the CV and the Cloud Agent during the remote attestation process; the changes made to the original implementation are highlighted in red. (1) The CV requests a new IR at the Cloud Agent by sending a GET request to the `/quotes/integrity` URI exposed by the Cloud Agent, comprising the following query parameters:

- *nonce*, a string of 20 alphanumeric characters used by the CV to verify the freshness of the quote received from the Cloud Agent;
- *mask*, a hexadecimal number used as bitmap to specify the PCRs that have to be included in the quote, with the exception of PCR 16, which must not be specified in the mask because it is always added to the quote since it contains the extension of the NK_{pub} key;
- *partial*, a boolean that specifies if the CV needs of the NK_{pub} ephemeral key generated by the Cloud Agent and used for encrypting the V share during the bootstrap key derivation protocol.

Upon receiving an attestation request, (2) the Cloud Agent requests a new quote from the TPM. In the example represented in figure 6.8, the hash algorithm configured in the Cloud Agent is `sha256` and the PCRs requested by the CV have indices 0, 1, 2 and 10; when the Cloud Agent requests the quote to the TPM, it specifies the PCRs with indices 0, 1, 2 and 16 belonging to the SHA-256 bank and PCR 10 (the IMA PCR) belonging to the SHA-1 bank. Then, the Cloud Agent composes the Integrity Report specifying:

- the TPM quote;
- the hash, encrypting and signing algorithms;
- (3) the contents of the entire IMA ML, if the *mask* sent by the CV specifies the IMA PCR;
- (4) the contents of the entire Measured Boot ML, if the *mask* sent by the CV specifies PCR 0.

The changes introduced in this procedure concern the IMA PCR contained in the quote and the IMA ML sent to the CV. Regarding IMA PCR, the following behaviour was introduced: the IMA PCR has to belong to the same bank chosen for the other PCRs, according to the configuration of the Cloud Agent. Regarding the IMA ML, an optimization was introduced that allows the CV to request only the portion of the IMA ML not yet attested, instead of the entire ML. To that end, a new query parameter, named `ml_seek`, was added to the attestation request; this parameter indicates the offset starting from which the CV requires the IMA ML contents, which correspond to the portion of the file not yet attested. This is possible because the CV, at each attestation cycle, memorizes the IMA PCR contained in the quote and the offset where the IMA ML matched the PCR; at the next attestation cycle, the CV sends the recorded offset in the `ml_seek` parameter and uses the IMA PCR of the previous attestation cycle as initialization value for checking the integrity of the portion of ML received. This optimization is significant when the time for reading and sending the IMA ML is comparable to the time needed to create the TPM quote.

6.4.3 Validating the Integrity Report

The IR validation process performed by the CV goes through several stages. First of all, the CV checks that the IR received from the Cloud Agent contains a quote and that the algorithms used for creating the quote are among those accepted. In particular, when the Tenant registers a new Cloud Agent in the CV, it also sends three lists containing the accepted algorithms for hash, encryption and signing. If the Cloud Agent sent an IR without quote or containing unacceptable algorithms, the validation process stops. Passed these first checks, the CV proceeds to validate the quote, verifying that:

- the quote is authentic by checking the signature with the AIK_{pub} retrieved from the Registrar;
- the quote is fresh, that is, it has been computed with the *nonce* provided in the IR request.

If these steps are passed, the CV examines the PCRs contained in the quote one by one, in order to establish the trustworthiness of host system and containers:

- if the PCR is the `TPM_DATA_PCR` (by default PCR 16), the CV checks that its content corresponds to the extension of the NK_{pub} key received from the Cloud Agent;
- if the PCR is the `IMA_PCR`, the CV checks that the IR contains the IMA ML and starts its validation process, described later;
- if the PCR belongs to the `MEASUREDBOOT_PCRS` list (PCRs with indices between 0 and 15 except IMA PCR) and if the CV received a Measured Boot policy from the Tenant, then it verifies that the Cloud Agent sent the Measured Boot ML (the content of the `/sys/kernel/security/tpm0/binary_bios_measurements` file) and validates it based on the Measured Boot policy;
- if the PCR does not fall in the previous categories, the CV verifies that its value is in the whitelist associated to that PCR and contained in the TPM policy.

Finally, the CV checks that the quote contains all the PCRs specified in the TPM policy. If one of the previous checks fails, it means that the host system is untrusted, so the CV stops the verification process, sets the operational state of the Cloud Agent to `INVALID_QUOTE`, notifies

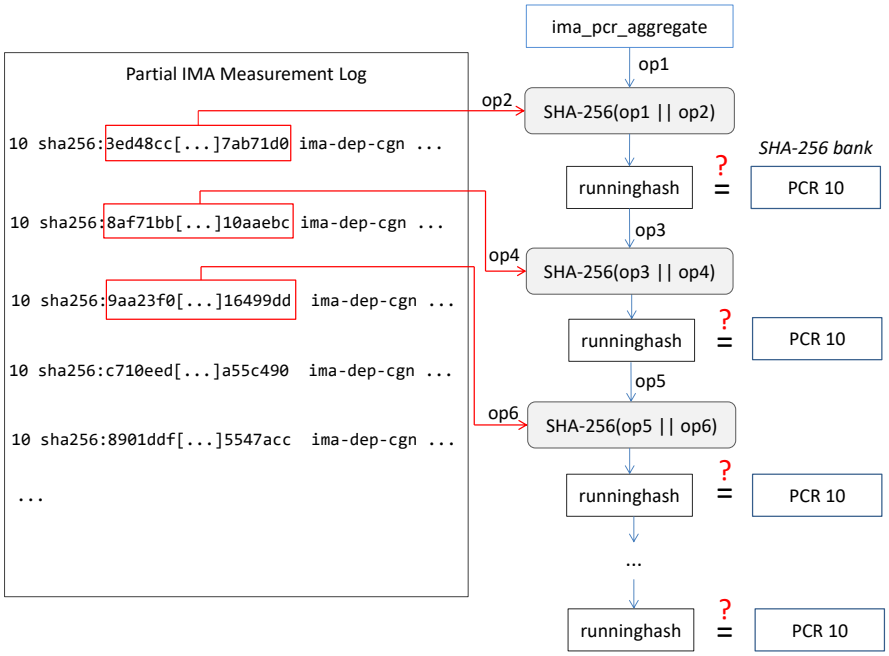

Figure 6.9. IMA ML integrity check for template-hash algorithm SHA-256

the host system's integrity verification failure via the Revocation Framework and suspends the periodic attestation. Instead, if all the previous checks are passed, it means that the host system is trusted, the Cloud Agent operational state remains to `GET_QUOTE` and the CV continues periodic attestation; however, the integrity state of one or more containers could be untrusted and this depends on the IMA ML validation process.

Validating the IMA Measurement Log

The process of validating the IMA ML consists of a series of checks performed for each ML entry. The CV splits the entry into fields by using the white space as separator, since the Cloud Agent sends the IMA ML in ASCII format. Then the CV, based on the template-name contained in the third field, verifies that the template-hash matches the template-fields of the entry by computing the digest on them with the hash algorithm received in the IR, as a result of the changes described above. If the computed digest is not equal to the template-hash contained in the entry, the CV records this error in an array that collects all errors detected during the ML verification process. Then, the CV computes the *extend* operation between the template-hash and the PCR aggregate, which is named `runninghash` in figure 6.9 and is constructed as the entries are examined, then the CV verifies if the result of the *extend* operation equals the IMA PCR received in the quote. During the first attestation of the Cloud Agent, the initialization value of the `runninghash` is an all-zero string since the CV receives the entire ML; in subsequent attestations, it is initialized with the value of the IMA PCR received in the previous attestation because the CV receives only the portion of ML not yet attested. If `runninghash` matches the IMA PCR received in the quote, this means that the ML contents are not tampered with up to that point. The figure 6.9 represents the sequence of template-hash extensions performed by the CV with hash algorithm SHA-256.

After verifying that the entry is not tampered with, the CV analyzes the contents of the template fields and, if the template is `ima-dep-cgn`, it performs the following verifications. The CV checks the `cgn` field in order to determine if the entry has to be associated to the host system or to a container. If the `cgn` field is a string composed of 64 hexadecimal characters, the CV analyzes the `dep` field and, if this contains the container runtime among the dependencies, the entry is considered associated to a container; otherwise, it is considered associated to the host system. If the `cgn` field is not a string of 64 hexadecimal characters, the CV immediately deduces

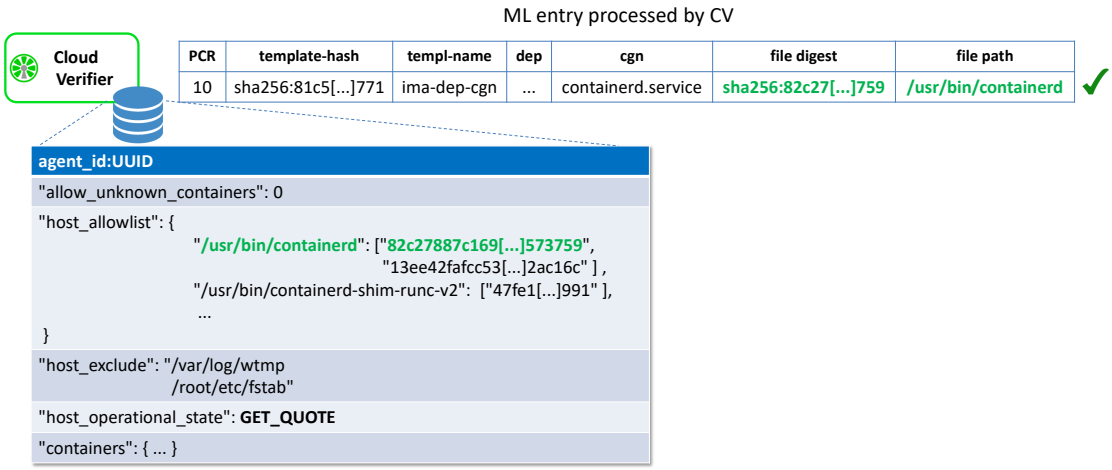

Figure 6.10. Representation of the CV's memory contents and a ML entry associated with the host sytem and evaluated as trusted by the CV.

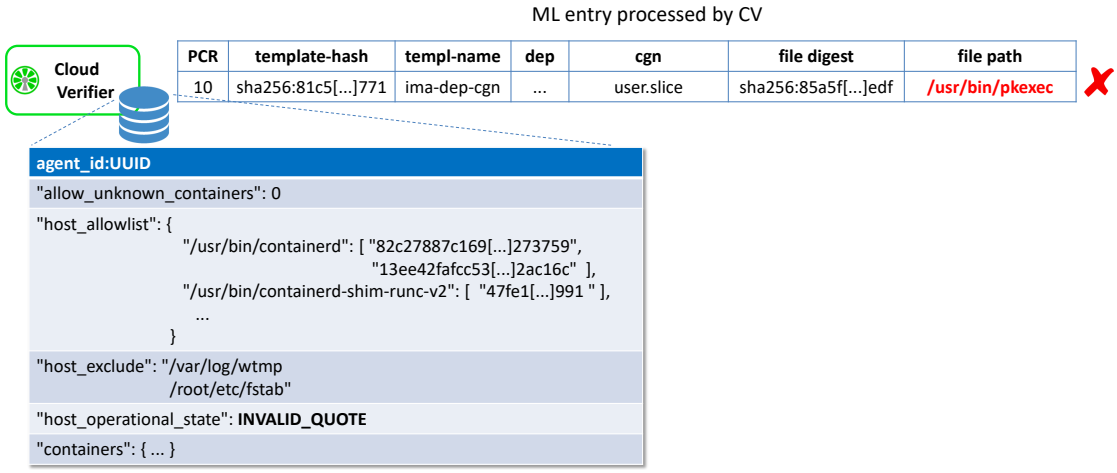

Figure 6.11. Representation of the CV's memory contents and a ML entry associated with the host sytem and evaluated as untrusted by the CV.

that the entry is associated to the host system.

Then the CV checks the filedata hash and file path fields for determining if the entry represents or not a trust event; for doing so, the CV uses the whitelist and exclude list associated to the host system or to the identified container. In the case of an entry belonging to the host system, the CV checks if the file path matches the exclude list of the host system (which is a regular expression representing all the file paths that do not need to be attested), in which case it doesn't perform other checks. If the file path doesn't match the exclude list, the CV checks if the whitelist (called `allowlist` in Keylime) related to the host system contains the file path and, in this case, if the file digest matches one of the trusted digests associated to that path. If the CV detects some errors in the entries related to the host system, it sets Cloud Agent's `operational_state` to `INVALID_QUOTE`, sends a message to the Revocation Notifier (described in 5.2.2) in order to communicate the change in the integrity state of that host system and stops the monitoring process; otherwise, the Cloud Agent's `operational_state` remains to `GET_QUOTE`.

The figures 6.10 and 6.11 represent two situations that can occur when the CV examines entries related to the host system. Figure 6.10 shows the case of an entry associated to the host system based on the `cgn` field; the CV evaluates it trusted because the file path and the file digest are both contained in the host system's whitelist. Instead, figure 6.11 shows the case of a host system entry resulted untrusted because the file path `/usr/bin/pkexec` is not contained in the

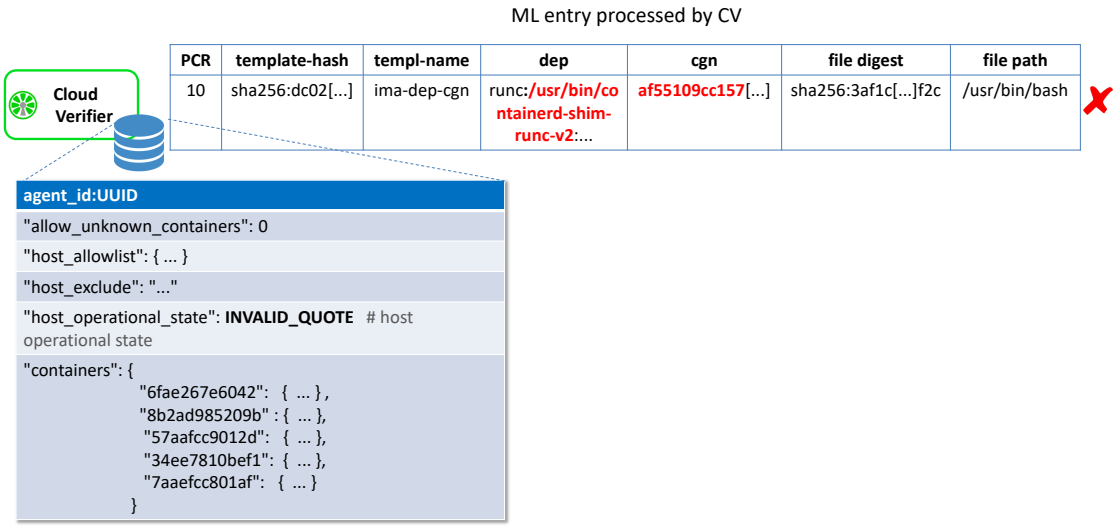

Figure 6.12. Representation of the CV's memory contents and a ML entry associated with an unregistered container and evaluated as untrusted by the CV.

host system's whitelist.

If the CV associates an entry to a container, because the `cgn` field has the format of a container full-ID and the `dep` filed contains the expected dependencies for a container, it takes the first 12 hexadecimal characters that represent the container ID and checks that a container with this ID is registered in the list of containers associated to that Cloud Agent. If the container ID is not found in the list and the `allow_unknown_containers` flag is set to `0`, the CV considers this event as an integrity failure and adds this container ID in the list of unknown containers detected. Instead, if the CV finds the container ID among those associated to the Cloud Agent, it goes on in the validation of the entry, firstly verifying if the file path matches the container's exclude list, then checking if file path and file digest are contained in the container's whitelist, in which case the entry is evaluated trusted. If a container's entry is evaluated untrusted, the CV sets the `operational_state` of that container to `UNTRUST`, then it sends a message to the Revocation Notifier related to the untrusted state of that container; however, the Cloud Agent's `operational_state` remains `GET_QUOTE` and the CV will continue the monitoring process for it. In this way, the CV is able to identify in a precise manner the portion of the platform that is compromised so that, when a container results untrusted, it is possible to stop only that specific container and replace it with a freshly created one, without the need to reset the whole platform.

The figures 6.12, 6.13 and 6.14 represent some situations that a CV can encounter while evaluating container entries. The figure 6.12 displays an entry related to a container with ID `af55109cc157` which is not registered in the list of containers associated to the Cloud Agent; the flag `allow_unknown_containers` set to `0` makes the presence of an unregistered container illegal, so the CV considers the host system's integrity status as untrusted and sets the Cloud Agent's operational state to `INVALID_QUOTE`. The figure 6.13 represents an entry of a container with ID `8b2ad985209b` evaluated trusted because the file digest field matches a trusted digest contained in the container's whitelist for that file path. Finally, the figure 6.14 represents an entry belonging to a container with ID `6fae267e6042` evaluated untrusted because the file digest field does not match any of trusted digests associated to the path `/usr/bin/bash` in the container's whitelist; this event does not change the integrity state of the host system, so the Cloud Agent operational state remains to `GET_QUOTE`.

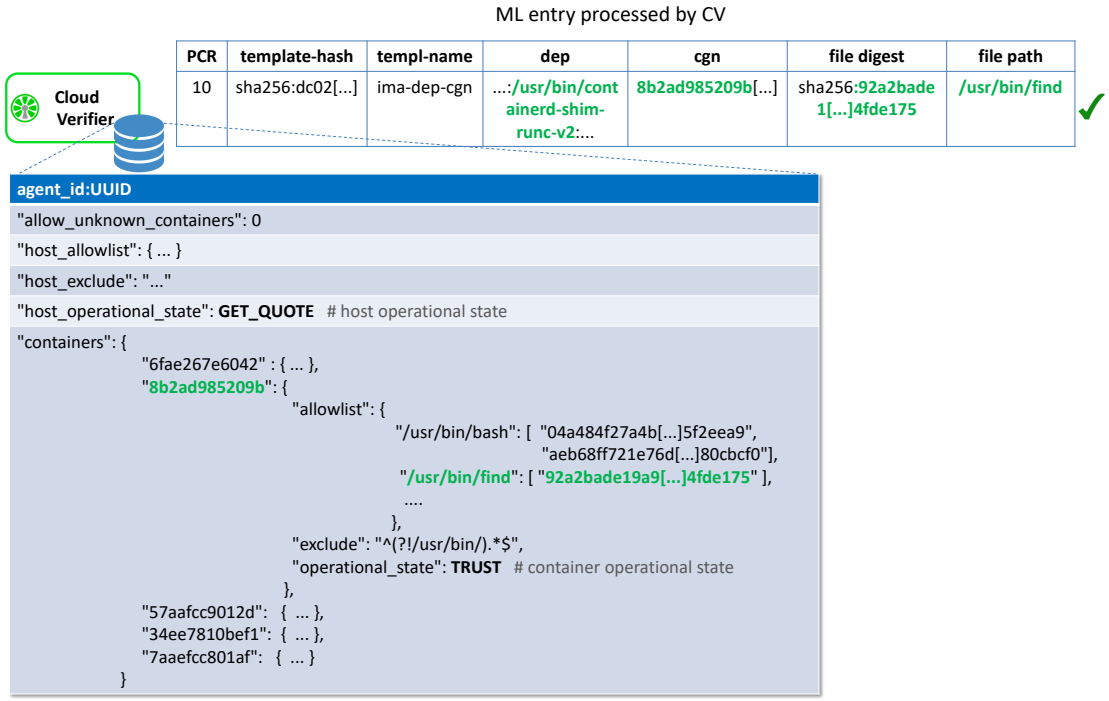

Figure 6.13. Representation of the CV's memory contents and a ML entry associated with a registered container and evaluated as trusted by the CV.

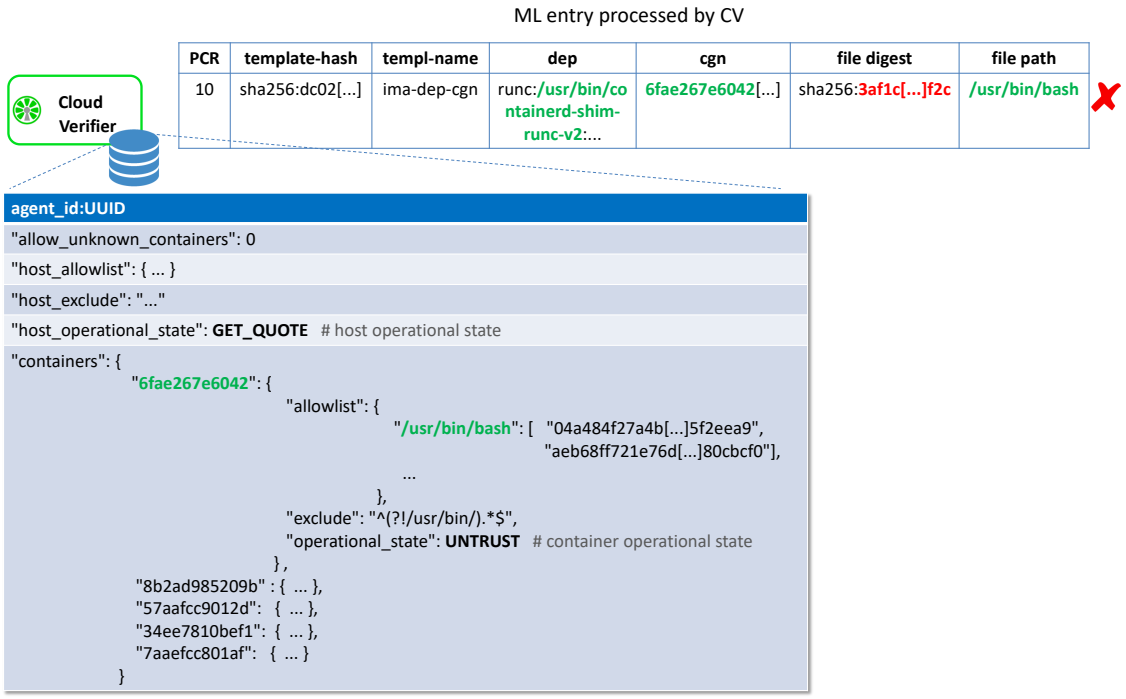

Figure 6.14. Representation of the CV's memory contents and a ML entry associated with a registered container and evaluated as untrusted by the CV.

Chapter 7

Trust Monitor

After having exposed the characteristics of the Network Functions Virtualization (NFV) paradigm in 7.1 and highlighted the need to monitor the integrity of a NFV network, the chapter focuses the attention on the Trust Monitor in 7.2, a monitoring entity developed with the aim of timely detect any tamper with a NFV platform tailored for the "Security-as-a-Service" scenario. Finally, a proposal of integration of the Trust Monitor with the Keylime framework is described in 7.3, for future use of this component in a generic cloud infrastructure.

7.1 Network Functions Virtualization

The traditional networking infrastructure requires, particularly at the edge of the network, to chain different dedicated hardware appliances which implement particular Network Functions (NFs), such as WAN accelerator, Firewall, Network Monitor, IDS, QoS. The Internet Service Providers (ISPs) offer added-value network services to their customers through the installation of these special-purpose appliances, which a network administrator has to manually configure and manage. This modus operandi involves a waste of time and resources, which translates in lack of flexibility and scalability of the service, long times for network innovation and great dependence on specialized hardware. In order to meet today's steadily growing demand of customers for new network services, the ISPs would have to continuously purchase and install new appliances and network administrators would have to quickly upgrade their skills, with a conspicuous increase in investment (CAPEX) and management (OPEX) costs. So, ISPs need of new solutions to innovate and optimize their network infrastructures, without passing costs on their customers or losing money.

Network Functions Virtualisation is an innovative network paradigm that proposes to substitute the traditional network infrastructures with highly virtualised platforms, where NFs are deployed as *Virtualised Network Functions* (VNFs), software implementations of the hardware appliances running on top of commodity servers. In this paradigm the NFs, that before were implemented in physical boxes, run now inside VMs or containers, enabling several advantages. The decoupling of NF software from the hardware facilitates the innovation process of both and shortens the development cycles, resulting in shorter time to market of new network services. Virtualization enables the automatic scaling of network services to adapt them according to current traffic; the scaling can be done in two different ways: scaling up/down (adding/removing computational resources to a NF) and scaling out/in (multiplying/reducing the number VMs that perform a given NF and adding a Load Balancer for splitting the traffic among them). VNFs can be deployed within a few tens of seconds in a completely dynamic and automatic way and they can be connected together in a completely programmable way, making it possible to easily differentiate the network services offered to customers. NFV simplifies the way in which dedicated services are offered to customers: it is just a matter of starting the VMs that implement the requested service, allocating them to that particular customer and ensuring that, with the appropriate OpenFlow rules (a protocol used with *Software-defined Networking* (SDN), which is

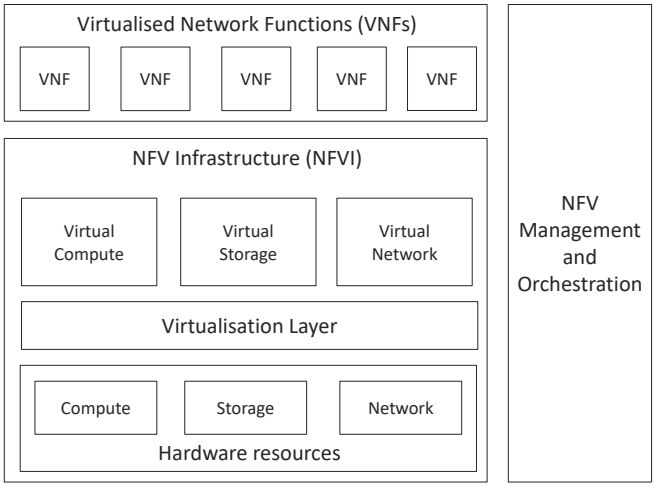

Figure 7.1. High-level NFV framework.

a complementary technology to NFV), only the traffic of that user ends up in those VMs. Moreover, the NFV paradigm facilitates the use of open source solutions, more advantageous than proprietary ones because they potentially have a lower cost and avoid the technology lock-in to a particular vendor. All of this translates into economic benefits for ISPs because it allows to reduce both management and upgrade costs of the ISP's infrastructure [45].

NFV may be considered as a special case of Cloud Computing because both paradigms enable the execution of applications and services on commodity servers on a large scale, the optimization of hardware resources and the service management through orchestration systems. However, a VNF has some peculiarities compared to a typical application running in the cloud. In the cloud computing world, a VM has CPU and memory as its most valuable resources, and produces little network traffic; in the NFV world, a VM containing the image of a NF has a low software complexity, so it consumes few CPU and memory resources, but it needs to dispose of an extremely high amount of traffic, of the order of tens of Gbps. For this reason, the pedestrian use of orchestrators borrowed from the cloud world would give suboptimal performance in the NFV environment: cloud computing orchestrators allocate VMs on the basis of the computational resources available on the physical servers, NFV orchestrators have to allocate VMs in a way that minimizes the amount of network traffic generated in the data center.

In order to facilitate the interoperability of NFV components and to allow different stakeholders to integrate NFV in their infrastructures, the "European Telecommunications Standards Institute" (ETSI) consortium founded, since 2012, an "Industry Specification Group" (ISG) that works on the standardization of NFV technology, which is based on three fundamental functional blocks [46], represented in figure 7.1:

1. ***Virtualised Network Function***, the software implementation of a network function which is capable of running over the NFV infrastructure;
2. ***NFV Infrastructure*** (NFVI), including the set of the physical resources and the software virtualization layer that support the execution of VNFs;
3. ***NFV Management and Orchestration*** (MANO), the set of components that entail the orchestration and lifecycle management of physical and software resources that enable virtualization, and the lifecycle management of VNFs.

Despite the numerous benefits offered by the NFV paradigm in terms of cost reduction, resource optimization and scalability, it introduces new security risks that could hinder its adoption in production environments. ETSI created the NFV-SEC Working Group (WG) with the aim of defining the NFV-specific security threats and the countermeasures to them. Their work [47]

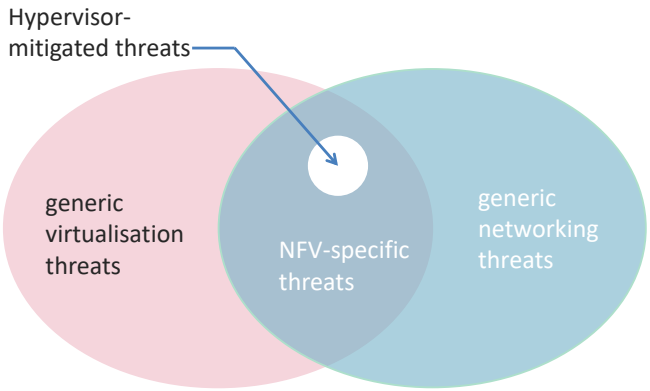

Figure 7.2. Visualisation of the NFV threat surface [47].

defined the NFV attack surface as the union of all generic threats of virtualization (e.g. memory leakage, interrupt isolation) with specific threats of physical network functions (e.g. flooding attacks, routing security); so, the new threats introduced by NFV are due to the combination of virtualization technology with networking, as shown on figure 7.2. However, it should be considered that virtualization can introduce security benefits as the hypervisor, thanks to introspection and other techniques, allows to eliminate or mitigate threats typical of physical network functions; this is represented in the figure by the presence of a "hole" in the intersection area. It follows that, in order to bring a NFV platform to a level of security sufficient to be deployed, it is necessary "to shrink" the intersection area and "to widen" that of the benefits provided by the hypervisor as much as possible.

The NFV-SEC WG listed the potential areas of concern regarding NFV-specific threats and stated the good rules to follow in order to counter the risk of attacks. One of the aspects analyzed with particular attention concerns the need to establish trust in all components of the NFV architecture, proposing Trusted Computing as the enabling technology to establish the trust in the network infrastructure. More specifically, NFVI nodes shall establish *hardware-based RoTs* for allowing the attestation process, which shall verify the integrity of *Network Services* (NSs) (logical entities comprising a chain of VNFs), NFVI computing nodes and network nodes (constituted by the SDN switches and the SDN controller). In order to realize that, NFV-SEC WG proposed the definition of a specific entity, i.e. the Trust Manager [48], containing all the trust logic for the entire NFV deployment, to sit in the MANO administrative domain as a long-lived entity. The availability of the Trust Manager gives great benefits to the NFV deployment:

- it allows a semplification of the logic of other entities within the deployment;
- it provides information through the different layers of the architecture;
- it interfaces the different administrative domains and operators;
- it acts as trusted repository for VNF packages and vendors;
- it provides historical data on entities that are longer-lived than the entity that wants establish trust on it.

However, it should be carefully designed and shielded because it becomes a single point of failure and a single point of attack.

7.1.1 SECaaS

ETSI described a list of use cases for the NFV paradigm, among which the *Security-as-a-Service* (SECaaS) use case. As online incidents and cybercrime are constantly growing and evolving, users and enterprises need to protect themselves from attacks and cyber threats. However the

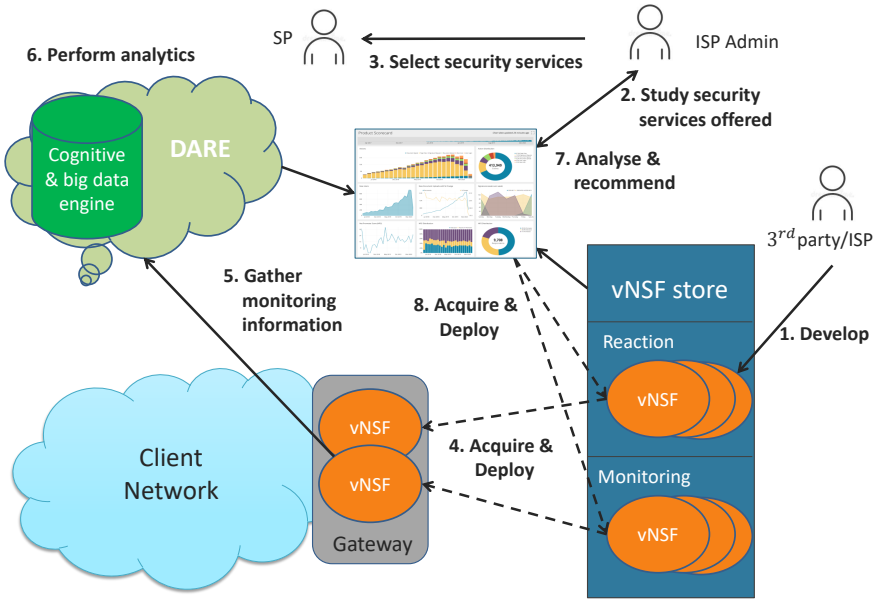

Figure 7.3. SecaaS based on NFV [47].

continuous updating of cybersecurity and cyber-defence techniques is a too expensive and complex process to be done effectively by the individual consumer or organization and usually it is not fast enough to counter the ever new types of attacks. This situation will worsen in the future due to several reasons: availability of ultrabroadband for residential customers, that can be used as DDoS tools; massive deployments of IoT devices which, for their characteristics of heterogeneity, low cost and lack of updates, can be exploited to access sensitive resources and data; an ever increasing Internet dependency of small enterprises which, not having enough resources to invest in cyber security, become perfect victims for cybercrime [49].

The NFV technology can provide solutions to this situation through a special type of VNF, the *Virtual Network Security Function* (vNSF), so it can be conveniently used to offer enhanced SECaaS services with the following benefits: dynamic and tailored response for security threats, automatic resources scalability, security data gathering and monitoring for intelligent analysis and remediation. The figure 7.3 represents this use case, where the actors are: the ISP customers, that require security services; the ISP operators, that provide network services; vNSF developers, that develop and publish the vNSF and can work for an internal ISP department or for third-party companies. The main components that constitute the SecaaS scenario are:

- *vNSFs*, security-oriented VNFs that can monitor the network (e.g., network probes, event generators, honeypots) or act on the network (e.g., firewalls);
- *Data Analysis and Remediation Engine (DARE)*, a central information-driven engine that processes data deriving from the vNSFs in order to discover current malicious behaviours and, by means of threat monitoring and cognitive intelligence techniques, takes decisions based on a holistic vision of the network;
- *vNSF store*, a logically centralized repository and catalogue used for adveritsing, browsing, selecting and trading vNSFs;
- *vNSF orchestrator*, a functionality associated to the NFV MANO stack, with the purpose of orchestrating security policies, managing the vNSFs and controlling their lifecycles;
- *Dashboard*, a graphic interface that allows the ISP customers or the ISP admin to interact with the NFV platform, providing different views on it depending on the user;
- *vNSF and infrastructure attestation*, an integrity monitoring process that proves to the ISP customers that the selected security service is trustworthy.

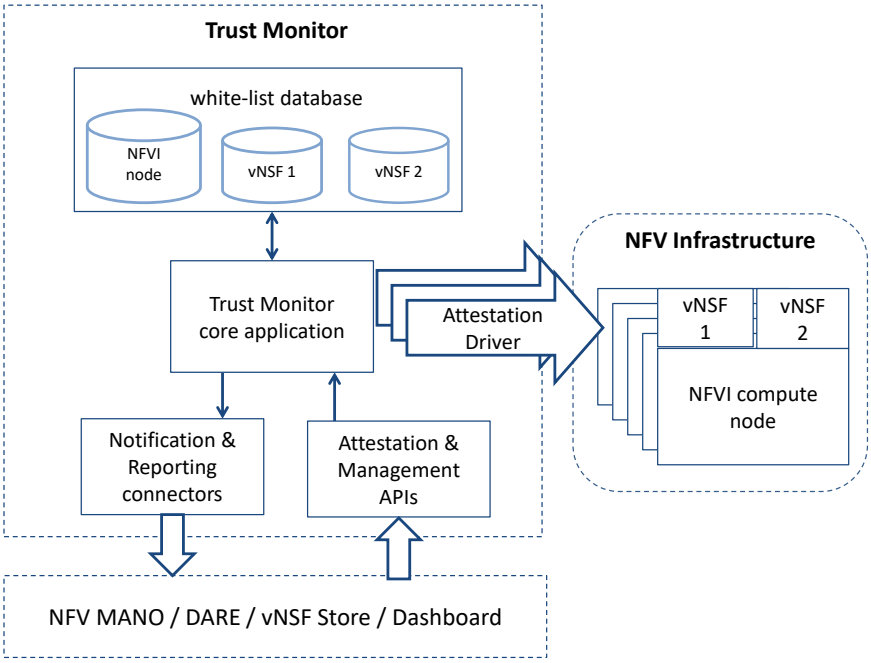

Figure 7.4. Trust Monitor architecture [45].

7.2 Trust Monitor for SECaaS use case

The *Trust Monitor* (TM) is a monitoring entity of a NFV platform and has the aim of providing remote attestation of both NFVI and VNFs in a SECaaS scenario, enabling the assurance that the deployed security services are trustworthy. It can be considered as an implementation of the Trust Manager entity proposed by ETSI [48]. The TM is not intended as an isolated component in the NFV platform but as an entity that cooperates with the other entities defined in the NFV SECaaS use case, described in 7.1.1. It was developed by the TORSEC research group of "Polytechnic of Turin" as part of the SHIELD project, funded by the European Union's Horizon 2020, and presented in 2019 at the "IEEE Conference on Network Softwarization" [45]. The TM has been designed as a stand-alone component inserted in the MANO administrative domain and it is compliant with the ETSI NFV Trust Manager definition [48]. The definition of the TM as a stand-alone component rather than as integrated in a MANO entity allows integrity verification of other MANO entities as well, even if its focus is the integrity verification of NFVI and vNSFs only. This is because they are directly exposed on the external ISP network and therefore more vulnerable to potential threats from external attackers; instead, the MANO domain is considered shielded from the operator's public network, so it is supposed to be protected from external attackers.

7.2.1 TM architecture

The TM was conceived with a modular architecture, whose sub-components are shown in figure 7.4, and provides the following functionalities:

1. *integrity verification* of both NFVI compute nodes and vNSFs;
2. *notification and reporting* of integrity status information to external entities;
3. *audit of historical logs* about the integrity status information.

The TM achieves the *integrity verification* of heterogeneous hosts in the NFVI by means of several *Attestation Drivers*, developed as plugins, which allow to instantiate different remote attestation

workflows, depending on the type of host, each one with its specific verification logic. This makes the TM able to attest nodes with different architectures (e.g., x86, ARM) and with RoTs based on different technologies (e.g., hardware TPM, Intel SGX, AMD SEV but also virtualised TEEs, although these offer lesser assurance than hardware-based ones), avoiding the lock-in with a particular vendor. Currently, three attestation drivers have been developed for supporting *Open Attestation* (OAT), *Open Cloud Integrity Technology* (OpenCIT) and *Hewlett Packard Enterprise Switch* (HPESwitch) frameworks. OAT allows integrity verification of NFVI compute nodes and vNSFs running in Docker containers, by means of the DIVE technology (described in 4.2), but it supports TPM 1.2 only. OpenCIT can attest only NFVI compute nodes but it supports both TPM 1.2 and 2.0 specifications. HPESwitch, instead, is used for integrity verification of SDN switches and SDN controller.

The TM uses different sources as *Whitelist Database* for retrieving the trusted reference measurements of NFVI nodes and vNSFs. In particular, regarding the NFVI nodes, the TM considers a global whitelist of reference measurements for the Linux distribution adopted by each of them; regarding the vNSFs, it relies on the vNSF Store as the source of reference measurements, whose values are either provided by the vNSF developer or obtained via static analysis of the vNSF image. Authentication and authorization policies, as well as replication and high availability of data, are considered mandatory for the Whitelist Database configuration in a production-oriented deployment of the TM.

The TM is an entity integrated in the NFV architecture, so it reports attestation results to external entities and exposes *Attestation and Management APIs* that allow to register NFVI nodes and attest them. The *notification and reporting* functionalities are implemented via pluggable *Connectors*, that allow the TM to interact with the other entities defined in the SecaaS scenario:

- *vNSFO Connector* for querying the vNSF Orchestrator about the list of physical nodes present in the NFVI and the vNSFs running on them;
- *vNSF Store Connector* for retrieving the reference measurements of the vNSFs;
- *DARE Connector* for conveying information about the trust status of the infrastructure and for long-term storage of historical audit logs;
- *Dashboard Connector* for showing end-users (ISP customers or ISP administrator) the attestation results.

7.2.2 NFVI attestation workflow

This section describes the steps, illustrated in figure 7.5, that implement the attestation process of a NFVI infrastructure in the TM architecture.

1. The vNSF Orchestrator sends to the TM an attestation request, regarding the entire NFVI platform or a subset of its nodes; then, for each node to be attested, the TM retrieves the list of vNSFs running on it.
2. For each NFVI node, the TM initiates the RA process through the specific *Attestation Driver* associated to the node. In the case of a TPM-based Attestation Driver, the Attester sends back to the TM an integrity report containing the IMA ML and a TPM quote.
3. The TM queries the vNSF Store for retrieving the list of reference measurements related to the vNSFs to be attested and stores them in its Whitelist Database; then, it verifies the authenticity of the IR and compares the measurements contained in it with those stored in the Whitelist Database.
4. The TM sends a notification with the attestation result to the Dashboard, containing the level of trust of both NFVI nodes and vNSFs. So, the ISP administrator can be aware about the trust status of the entire NFV platform, while the ISP customer can know the trustworthiness of the purchased Network Services.
5. The TM forwards the entire attestation log to the DARE, which will use it for data analytics and for long-term storage of attestation results.

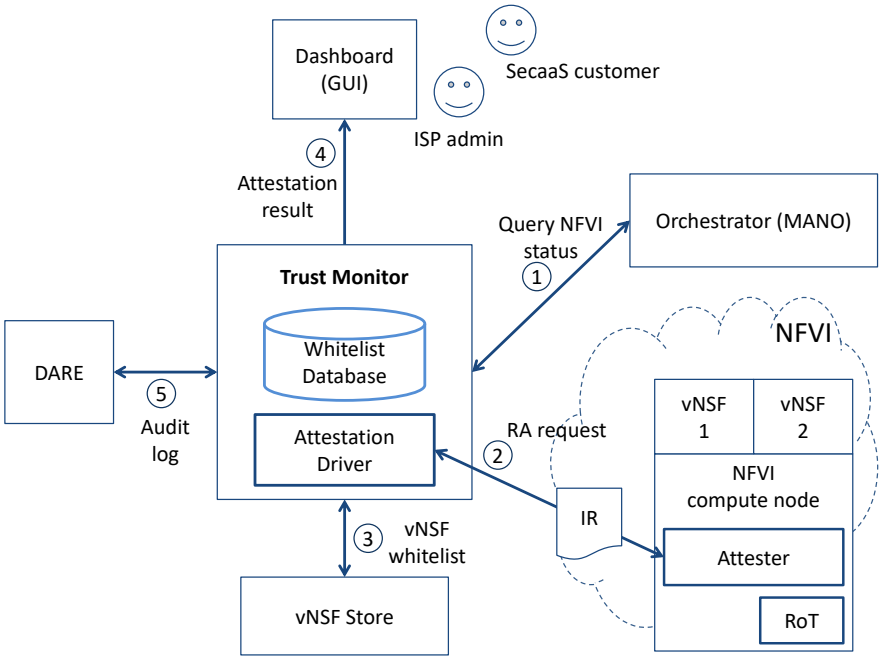

Figure 7.5. The Trust Monitor NFVI attestation process [45].

7.3 Trust Monitor 2.0

The TM described in the previous section is tailored for the SECaaS use case. However, its modular architecture makes it adaptable to heterogeneous deployment scenarios, since the TM core logic is decoupled from the SECaaS-specific workflow. So, in the next future, the designers plan to use this monitoring entity in a generic cloud deployment based on lightweight virtualization, updating it with new emerging technologies. With this in mind, the TM 2.0 is an entity capable of interacting with a container orchestrator, like Kubernetes, and attesting containers where generic cloud applications run. My contribution in this project was to create a new attestation driver for the integration of Keylime in the TM, and a Whitelists Web Service for the creation and management of host and container whitelists. The figure 7.6 represents the TM architecture with the components introduced with the thesis work.

The TM has been implemented as a set of microservices, where the TM core application has the central role; each of them is a Web Service developed in Python language, with a Dockerfile that allows its deployment through the Docker container engine; in addition, the Docker Compose tool was used to quickly and efficiently instantiate all sub-components of the TM and easily enable network interaction between them.

7.3.1 TM core application

The TM core application was developed with Django REST web framework and exposes APIs to register computational nodes of a cloud infrastructure, to attest them and to verify the correct configuration of all the TM's subcomponents. It manages an internal database, created automatically when the TM is started based on the classes defined in the `models.py` file. In particular, this file defines the `Host` class, which represents a physical node of the infrastructure for which it is possible to request the attestation procedure, and the `KnownDigest` class, which represents the digest of a custom software running on a physical node. For each class, the Django REST framework creates a table in the database and each instance of the class can be saved in the corresponding table. The digests stored in the KnownDigest table are used for creating the whitelist of a physical host, together with the reference measurements corresponding to the Linux distribution present on the host.

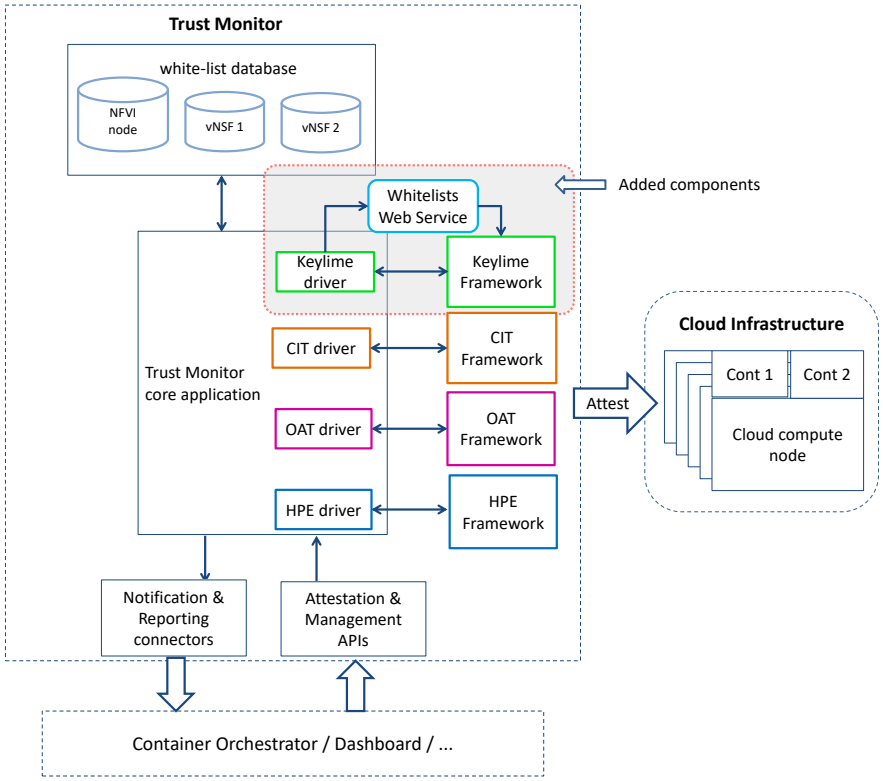

Figure 7.6. Trust Monitor architecture with added components.

The registration and the deletion of physical hosts within the TM takes place through the API `https://<trust_monitor_IP>/register_node/`, on which the following HTTP methods can be invoked: `GET` to retrieve the list of all the physical hosts registered in the TM; `POST` to register a new host inside the TM and in the corresponding attestation framework, specified in the request body; `DELETE` to remove a previously registered host from both the TM and the attestation framework. The integrity check of one or more hosts can be done through three different APIs:

- `https://<trust_monitor_IP>/attest node/` exposes the POST method to attest one or more nodes with their containers, whose identifiers are contained in the request body;
- `https://<trust_monitor_IP>/nfvi_attestation_info/` exposes the GET method to attest all nodes registered with the TM;
- `https://<trust_monitor_IP>/nfvi_pop_attestation_info/?node_id=...` exposes the GET method to attest only one node in the infrastructure, whose identifier is specified as query parameter.

The attestation process of each host and of its containers is performed through the specific attestation framework associated with it in the registration phase. The contents of the KnownDigest table can be managed through the API `https://<trust_monitor_IP>/known_digests/`, which exposes three HTTP methods: `GET` for retrieving the list of all the digests contained in the table, `POST` for adding a new digest to the table, `DELETE` for removing a digest from the table.

The TM core application exposes also management and audit APIs:

- `https://<trust_monitor_IP>/status/` allows to verify, through a GET request, the correct configuration and the active status of all the sub-components within the TM architecture, in particular attestation drivers, connectors and databases;
- `https://<trust_monitor_IP>/audit/` allows to retrieve, through a POST request, the audit log of the attestations performed on a specific node.

Further details regarding the REST APIs exposed by the TM are described in appendix F.

7.3.2 Whitelists Web Service

The Whitelists Web Service is a new sub-component of the TM architecture, created with the aim of realizing the logic concerning the management of the whitelists of hosts and containers within a single entity, that interfaces with the other TM's components through REST APIs. The original implementation of the Whitelist Database relies on three sources:

- the Cassandra database, represented in figure 7.6 by the "NFVI node" DB, containing a unified global whitelist related to the Linux distribution present in all the physical hosts of the infrastructure;
- the KnownDigest table of the internal database, containing the digests of the proprietary software running on the physical hosts;
- the vNSF Store containing the whitelists for the containers, accessed through the vNSF Store Connector; once retrieved from the connector, the container whitelists are stored in a Redis database for being used during attestation.

The presence of various sources of information, necessary for the realization of the whitelist of hosts and containers, however, makes the core logic of the TM more complex; hence the idea of adding a new component that would centralize this complexity in a single point, offering simplified access to the whitelists of the various entities to be attested to the sub-components that need them. The Whitelists Web Service offers the ability to create specific whitelists for each host in the infrastructure, instead of having a single global whitelist for all hosts; this makes the whitelists more consistent with the specific software configuration present on the host to be attested. This service is also able to periodically update the contents of its database with respect to the software updates published in the reference mirrors of the supported Linux distributions; the frequency with which the update can be performed can be set through its configuration file. Moreover, while the Cassandra database populating software provides whitelists only for the sha-1 algorithm, this service provides whitelists for hosts and containers with various hash algorithms: sha1, sha256, sha384, sha512.

The service exposes two sets of APIs for managing whitelists, one relating to physical hosts, the other relating to containers. As for the first group, it provides three APIs:

- `/hosts/{host_id}` exposes the methods: `PUT` to create / update the whitelist relative to the host with the identifier specified in the URI, created on the basis of the host configuration (packages installed and platform architecture) and the selected hash algorithm; `GET` to retrieve the host's whitelist;
- `/hosts/{host_id}/known_digest` exposes the `POST` method to add in the whitelist of the host, whose identifier is specified in the URI, a custom digest contained in the body of the request;
- `/packages/{package_name}` exposes the methods: `PUT` to update / download a package, whose name is specified in the URI, with version and platform architecture indicated in the body of the request; `GET` to retrieve package information for all versions and platform architectures currently contained in the database.

The API concerning the container whitelists are:

- `/images/{image_id}` exposes the methods: `PUT` to create / update the whitelist corresponding to a container image, which can be "pulled" or "build" depending on what is specified in the body of the request; `GET` to retrieve information relating to the whitelist of the image and the layers of which it is made up;

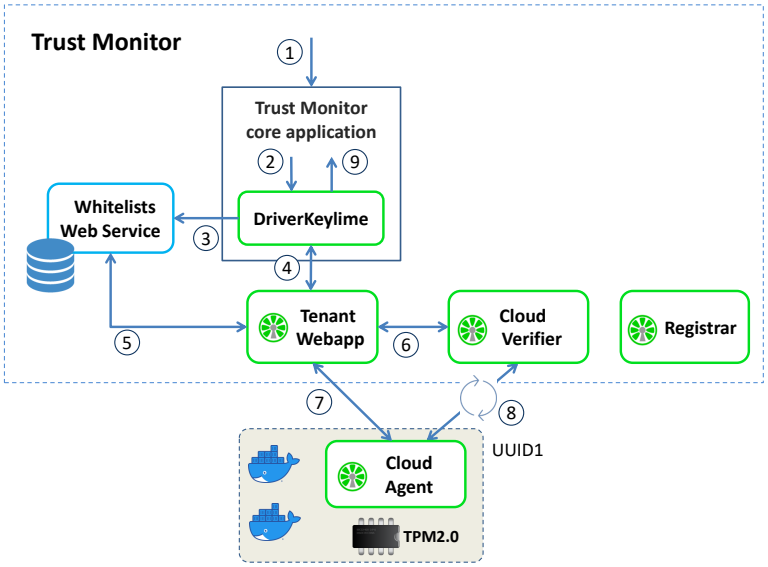

Figure 7.7. registerNode() workflow with DriverKeylime.

- `/containers/{container_id}` exposes the methods: `PUT` to create / update the whitelist corresponding to a container whose identifier is indicated in the URI, based on the whitelist of a container image and the selected hash algorithm; `GET` to retrieve the container's whitelist.

Further details on the REST APIs exposed by the Whitelists Web Service are illustrated in the appendix E.

7.3.3 Keylime Attestation Driver

A TM Attestation Driver is a specific implementation of a generic integrity verification interface which exports four methods:

- `registerNode()` for registering a new host in the attestation framework;
- `pollHost()` for checking the integrity state of a host through the attestation framework;
- `getStatus()` for getting the status of the attestation framework, that is, if it has been configured and is active;
- `deleteNode()` for removing a host previously registered in the attestation framework.

In order to enable the TM to use Keylime, it was necessary to create a new Attestation Driver which implements the four methods mentioned above, whose workflows are described in the following sections.

Registering a new host

The workflow for registering a new host with the Keylime Attestation Driver is represented in figure 7.7. (1) The process starts when the TM receives a POST request to the URI `/register_node/`, with body containing the host's UUID, the IP address, the "distribution" (used for specifying the host architecture and the list of installed packages), the name of the Attestation Driver (in this case "Keylime"), the list of containers running on the host and a parameter that allows/does not allow the presence, in the host, of other containers besides those listed. Each container is specified

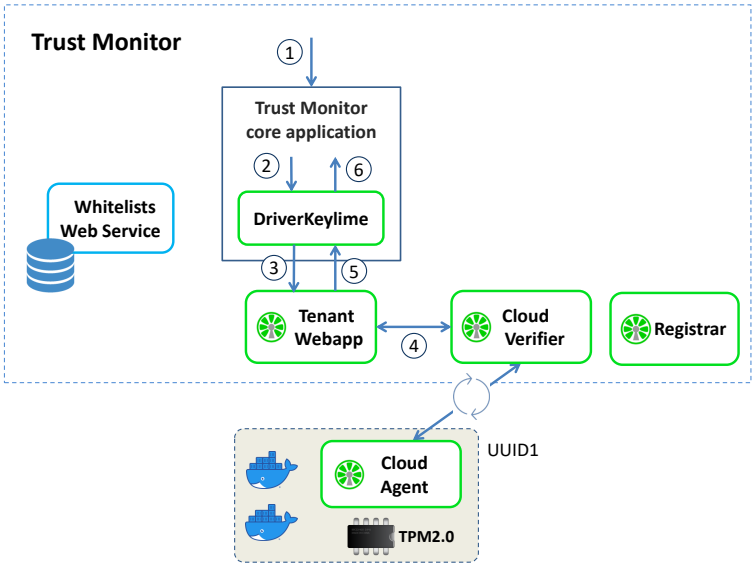

Figure 7.8. pollHost() work-flow with DriverKeylime.

through its container ID and its image ID, which identifies the whitelist associated to the container image. (2) The TM core application invokes the registerNode() method of the Attestation Driver specified in the POST request, in this case the DriverKeylime. (3) The DriverKeylime asks the Whitelists Web Service to create the whitelist corresponding to the host, based on the "distribution" parameter specified in the previous POST request, and the whitelists of each container, based on its "image-ID". (4) The DriverKeylime registers the new host inside the Keylime framework by sending a POST request to the URI `/v2/agents/{agent_uuid}` exposed by the Keylime Tenant Webapp, specifying the URLs through which host and container whitelists can be downloaded. (5) The Keylime Tenant Webapp downloads from the Whitelists Web Service the whitelists for host and containers, (6) then it registers the new host in the Keylime Verifier and (7) (8) the Three Party Bootstrap Key Derivation Protocol takes place, as described in section 5.2.2. After registration, the Keylime Verifier begins the periodic attestation of the host system and the registered containers, as described in 6.4.2 and 6.4.3 sections. (9) If the registration procedure in the Keylime framework is successful, the TM core application registers the new host in its internal database.

Checking the integrity state of host and containers

The workflow related to the integrity check of a host associated to the DriverKeylime is illustrated in figure 7.8. (1) The TM core application receives a POST request to the URI `/attest_node/`, with body that contains a list of nodes to be attested, among which at least one has "Keylime" as Attestation Driver. (2) For each of these, the TM invokes the pollHost() method of the DriverKeylime, (3) which performs a GET request to the URI `/v2/agents/{agent_uuid}` exposed by the Keylime Tenant Webapp, in order to retrieve the current integrity status of the node. (4) In turn, the Tenant Webapp performs a GET request to the URI `/v2/agents/{agent_uuid}` exposed by the Keylime Verifier, which sends back the integrity status of host and containers as it results from the last remote attestation process performed for that host. (5) The Tenant Webapp responds to the DriverKeylime with the information received by the Verifier; then, (6) the DriverKeylime creates a JSON object which represent the attestation log and returns it to the TM core application.

Getting the status of the Keylime framework

The TM core application provides also the GET `/status/` REST API for checking the status of all the components inside the TM architecture, in particular the Attestation Drivers, the Connectors and the Databases configured in it. Regarding the Attestation Drivers, the TM invokes the getStatus() method on each of them. The response of this method is a JSON object that specifies if the attestation framework has been configured and is active. In the DriverKeylime, the getStatus() implementation checks if the IP address of the Keylime Tenant Webapp has been configured in the TM's setting file and if the Keylime framework is active by sending a request to one of the REST APIs exposed by the Keylime Tenant Webapp.

Deleting a host from Keylime

When a node has to be removed from the cloud infrastructure, the TM receives a DELETE request to the URI `/register_node/`, which causes the node to be deleted from the TM internal database. In order to propagate the host remotion to the attestation frameworks, a new method, deleteNode(), has been added to the Attestation Driver's interface. The implementation of deleteNode() in DriverKeylime sends a DELETE request to the URI `/v2/agents/{agent_uuid}`, exposed by the Keylime Tenant Webapp which in turns sends the same request to the Keylime Verifier. This stops the periodic remote attestation of the host and removes it from the Verifier database.

Chapter 8

Testing

The chapter presents the results of the tests performed on the Keylime framework, modified in order to perform container attestation. In particular, functional tests were performed to verify the correct behavior of the attestation process with the Trust Monitor and Keylime frameworks 8.2, and performance tests were executed to evaluate the latency times and the resources consumption in the attestation process 8.3.

8.1 Testbed

To evaluate the functioning and performance of the proposed solution, the testbed was set up as follows:

- an attester machine having an Intel Core i5-5300U CPU (2 core, 4 threads), 16 GB of RAM and a discrete Infineon TPM 2.0 chip, running Ubuntu Server 20.04 LTS with a custom Linux kernel, based on version 5.12, containing the modification to the IMA module described in 6.3; Docker Consumer Edition version 20.10.8 and the Keylime Agent, modified as described in 6.4.2, have been installed and are running on it;
- a verifier machine having an Intel Core i7-3520M CPU (2core, 4 threads) and 8 GB of RAM, with Ubuntu Dsktop 20.04 LTS operative system on which Trust Monitor, Keylime Verifier, Keylime Registrar, Keylime Tenant Webapp and Whitelists Web Service have been installed and are running.

The appendix A describes in detail the installation and configuration of the testbed.

8.2 Functional tests

The purpose of functional tests is to verify whether the software implementation of the proposed solution meets the requirements. Before proceeding with the exposition of the tests, let us recall some assumptions used by the *Keylime Verifier* for attesting a physical host on which containers are running:

- a *physical host* is considered *trusted* if it performed a trusted boot, so the PCRs related to the boot phase match the reference values, and if the measurements of the files accessed at runtime are present in the host's whitelist, regardless of the state of the containers running on it; the "allowUnknownContainers" flag associated with the host allows to evaluate differently the case in which not-registered containers are detected on it: if it is "true", their presence does not affect the integrity state of the host, otherwise the host is considered *untrusted*;

- a *container* is considered *trusted* if the measurements of the files accessed after its instantiation are present in the whitelist associated to it.

The *Trust Monitor* is currently tailored for the integrity evaluation of a NFV infrastructure, which is the set of physical and virtual resources deployed in a network platform. It will evaluate the *NFVI* platform *trusted* only if all its components, both physical and virtual, are trusted.

8.2.1 Tests with trusted platform

What we expect from this test is that, if we have the attesting system and all its containers with a software configuration compliant with their whitelists, the Keylime Verifier will evaluate "trusted" physical host and containers and consequently the TM will evaluate "trusted" the NFVI platform.

On the attesting system, the Keylime Agent has been configured with uuid "UUID1" and two Docker containers have been deployed, one launched with base image `ubuntu`:

```
# docker run -it ubuntu /bin/bash
```

and the other launched with base image `nginx`:

```
# docker run -d -p 30000:80 nginx
```

Their container IDs are `237774a3deb4` and `4e2cacb35b8d` respectively. The attesting system has been registered with the TM by sending a POST request, with body represented below, at `https://<TM_IP_address>/register_node/`:

```
{
   "hostName": "UUID1",
   "address": "192.168.1.50",
   "distribution": "{
                       \"architecture\":\"amd64\",
                       \"hash_algorithms\":\"sha256\",
                       \"packages_list\":\"acl 2.2.53-6 amd64\n
                                           adduser 3.118ubuntu2 all\n
                                           ..."
                   }",
    "driver": "Keylime",
    "containers": "4e2cacb35b8d nginx\n237774a3deb4 ubuntu",
    "allowUnknownContainers": 0
}
```

See section 7.3.3 for a description of the registration work-flow. Completed the registration phase, the Keylime Verifier begins the periodic attestation.

When the TM receives the request to attest "UUID1", either through a GET request at `https://<TM_IP_address>/nfvi_pop_attestation_info/?node_id=UUID1` or via a POST request at `https://<TM_IP_address>/attest_node/` with body as below:

```
{
   "node_list": [ { "node": "UUID1" } ]
}
```

it sends back the following response:

```
{
   "hosts": [
             {
               "node": "UUID1",
               "status": 0,
               "time": "2021-09-03 20:03:41.129083 +0000 UTC",
               "remediation": {
                                 "terminate": false,
                                 "isolate": false
                               },
```

```
                "vnsfs": [
                          {
                            "container": "4e2cacb35b8d",
                            "vnsfr_id": "",
                            "vnsfd_id": "",
                            "remediation": {
                                             "terminate": false,
                                             "isolate": false
                                           },
                            "trust": true,
                            "ns_id": ""
                          },
                          {
                            "container": "237774a3deb4",
                            "vnsfr_id": "",
                            "vnsfd_id": "",
                            "remediation": {
                                             "terminate": false,
                                             "isolate": false
                                           },
                            "trust": true,
                            "ns_id": ""
                          }
                        ],
                "trust": true,
                "driver": "Keylime",
                "extra_info": { ... }
            }
        ],
        "sdn": [],
        "trust": true,
        "vtime": "2021-09-03 20:03:41.130289 +0000 UTC"
    }
```


where:

- `"hosts"` contains the list of the attested compute nodes;
- `"trust"` indicates the trust level assigned by the TM to the NFVI infrastructure, based on the attestation process executed by the Keylime Verifier on the platform components;
- `"vtime"` records the moment in which the attestation process was performed.

The parameters that describe the result of the attestation process for a host are:

- `"node"` is the uuid of the Keylime Agent;
- `"status"` indicates if the integrity process successfully completed;
- `"time"` is the time in which the attestation process took place;
- `"remediation"` specifies the actions to be performed on the node after the attestation process;
- `"vnsfs"` contains the list of containers running on the host, each of them indicating the container ID and the trust level;
- `"trust"` specifies how the Keylime Verifier evaluated the trustworthiness of the physical platform;
- `"driver"` indicates the attestation driver used to attest the host;
- `"extra_info"` contains additional details which the current implementation of the Keylime driver does not evaluate.

Since this object was defined to describe an NFV platform, some fields (e.g., `"sdn"`, `"vnsfr_id"`, `"vnsfd_id"`) were not considered in the above list because they do not assume a meaningful value for the context of this test.

As we can see form the TM response, the Keylime Verifier correctly evaluated the integrity state of the host "UUID1" and its containers and the TM correctly assessed the trust level of the entire platform, inferring that it is trusted in accordance with the fact that both the physical host and the containers are trusted.

8.2.2 Tests with untrusted platform

Now, let us execute inside the "ubuntu" container a software that is not part of its base image, so it is not present in its whitelist; since we executed this container in "interactive" mode, we can download, from its shall, the "nano" text editor and create a new file with it, so that this new software is executed and consequently a measurement related to it is added inside the IMA ML:

```
root@237774a3deb4:/# apt update
root@237774a3deb4:/# apt install nano
root@237774a3deb4:/# nano new_file.txt
```

What we expect is that:

- the Keylime Verifier evaluates the container `237774a3deb4` as "untrusted", while the container `4e2cacb35b8d` and the physical host, whose software configurations have not been tampered with, as "trusted";
- the TM evaluates the entire platform as "untrusted" since one of its components is "untrusted".

When we send an attestation request for the host "UUID1" to the TM, it sends back the following response:


```
{
   "hosts": [
              {
                "node": "UUID1",
                "status": 0,
                "time": "2021-09-04 13:36:59.009288 +0000 UTC",
                "remediation": {
                                  "terminate": false,
                                  "isolate": false
                               },
                "vnsfs": [
                           {
                             "container": "4e2cacb35b8d",
                             "vnsfr_id": "",
                             "vnsfd_id": "",
                             "remediation": {
                                               "terminate": false,
                                               "isolate": false
                                            },
                             "trust": true,
                             "ns_id": ""
                           },
                           {
                             "container": "237774a3deb4",
                             "vnsfr_id": "",
                             "vnsfd_id": "",
                             "remediation": {
                                               "terminate": true,
                                               "isolate": true
                                            },
```

```
                            "trust": false,
                            "ns_id": ""
                        }
                    ],
                "trust": true,
                "driver": "Keylime",
                "extra_info": { ... }
            }
    ],
    "sdn": [],
    "trust": false,
    "vtime": "2021-09-04 13:36:59.010558 +0000 UTC"
}
```


As we can see from the JSON object, the Keylime Verifier and the TM correctly evaluated the trust level of platform components.

Now, let us create, inside the `/usr/bin/` directory of the host system, a simple "Hello World!" script, assign execution privileges to it and run it. What we expect is that the Keylime Verifier evaluates also the host system as "untrusted" and the TM evaluates the platform "untrusted" as well. The attestation result provided by the TM is shown below:


```
{
    "hosts": [
            {
                "node": "UUID1",
                "status": 0,
                "time": "2021-09-04 18:50:11.403360 +0000 UTC",
                "remediation": {
                                "terminate": true,
                                "isolate": true
                            },
                "vnsfs": [
                            {
                                "container": "4e2cacb35b8d",
                                "vnsfr_id": "",
                                "vnsfd_id": "",
                                "remediation": {
                                                "terminate": false,
                                                "isolate": false
                                            },
                                "trust": true,
                                "ns_id": ""
                            },
                            {
                                "container": "237774a3deb4",
                                "vnsfr_id": "",
                                "vnsfd_id": "",
                                "remediation": {
                                                "terminate": true,
                                                "isolate": true
                                            },
                                "trust": false,
                                "ns_id": ""
                            }
                        ],
                "trust": false,
                "driver": "Keylime",
                "extra_info": { ... }
            }
    ],
    "sdn": [],
    "trust": false,
    "vtime": "2021-09-04 18:50:11.404675 +0000 UTC"
}
```

The JSON object shows that the trustworthiness assessment performed by Keylime Verifier and TM is correct.

Finally, in order to test the correct detection by the Keylime Verifier of unregistered containers running on the host, we have to reboot the attesting system in order to reset the IMA ML, then we deploy two containers on it. Now we have to remove the host "UUID1" from the TM by sending a DELETE request at `https://<TM_IP_address>:443/register_node/` with body:

```
{
   "hostName": "UUID1"
}
```

Finally, we have to register the host with the TM again, this time specifying in the request body only one container. What we expect is that the Keylime Verifier detects the presence of an "unauthorized" container and evaluates the host system as "untrusted"; the TM consequently should evaluate the platform as "untrusted" as well. By sending an attestation request for the host "UUID1", the TM's response is as follows:

```
{
   "hosts": [
             {
               "node": "UUID1",
               "status": 0,
               "time": "2021-09-04 20:05:08.086768 +0000 UTC",
               "remediation": {
                                "terminate": true,
                                "isolate": true
                              },
               "vnsfs": [
                          {
                            "container": "37aba78dcc2b",
                            "vnsfr_id": "",
                            "vnsfd_id": "",
                            "remediation": {
                                             "terminate": false,
                                             "isolate": false
                                           },
                            "trust": true,
                            "ns_id": ""
                          }
                        ],
              "trust": false,
              "driver": "Keylime",
              "extra_info": { ... }
          }
    ],
   "sdn": [],
   "trust": false,
   "vtime": "2021-09-04 20:05:08.088090 +0000 UTC"
}
```

As it appears from the JSON object, also in this case Keylime Verifier and TM correctly evaluated the trust level of the attesting platform.

8.3 Performance tests

The performance evaluation of the proposed solution focused on three metrics: time taken by the Keylime Verifier to perform an attestation cycle; CPU utilisation and RAM consumption on the attesting platform, evaluated with and without the remote attestation process. The values relative to these metrics have been acquired as the number of containers instantiated on the attesting system increases, starting from 1 container up to a deployment scenario of 512 containers, all having nginx as image. Performance beyond 512 containers was not evaluated due to the limits

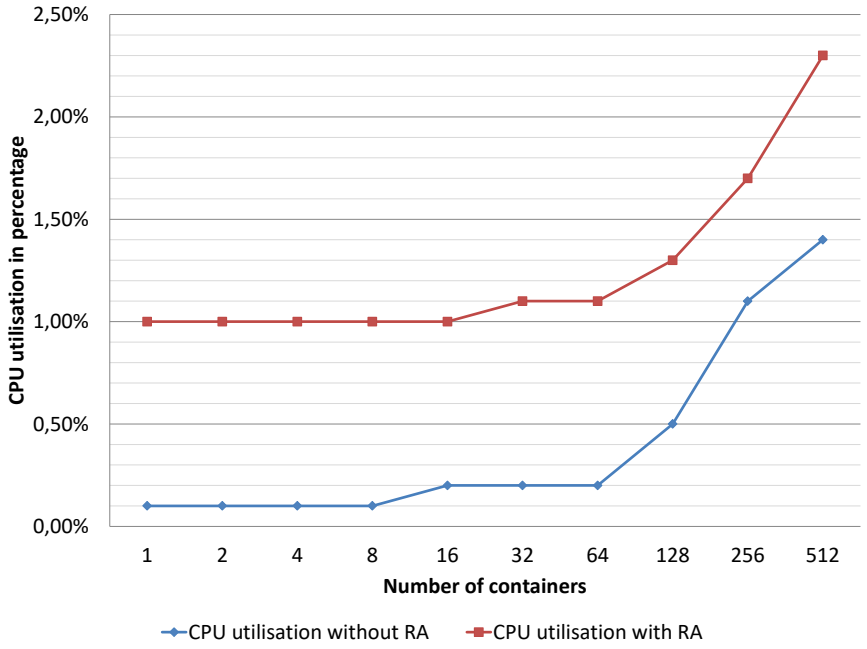

Figure 8.1. CPU consumption on the attester machine as the number of containers increases.

of the resources available on the attesting platform. CPU and RAM consumption were evaluated over a time window of 10 minutes, while the attestation time was evaluated on the average of 300 attestations performed for each group of containers taken into consideration, in order to get statistically meaningful results.

The figure 8.1 depicts the CPU utilisation in percentage on the attester, detected both when the periodic remote attestation is active and in absence of it, in order to test the CPU consumption penalty introduced by the Keylime Agent and the tpm2-tools used by it. The results show that the attestation process leads to a 1% increase in CPU usage, which remains almost constant at the increase of containers deployed on the attesting platform.

The figure 8.2 highlights that the presence of the attestation process has a negligible impact on RAM consumption since the graphs obtained from the data acquired on the attesting platform, with and without the attestation process, appear overlapping, regardless of the number of containers deployed on the platform.

Finally, the figure 8.3 shows the time required for the Keylime Verifier to complete the attestation process. The different contributions to the overall attestation time are detailed in the following equation:

$$T_{RA} = t_{IR\_req} + t_{quote} + t_{ML\_reading} + t_{IR\_sending} + t_{IR\_verification} \tag{8.1}$$

where:

- t_{IR_req} represents the time for sending and transferring an attestation request over the network; this contribution can be considered negligible;
- t_{quote} represents the time the TPM takes to create a quote;
- $t_{ML\_reading}$ is the time needed for the Keylime Agent to read the IMA ML file in order to insert it in the IR;
- $t_{IR\_sending}$ is the time needed for sending and transferring the IR over the network;
- $t_{IR\_verification}$ is the time the Keylime Verifier takes to authenticate the IR and verify the integrity of both physical host and containers.

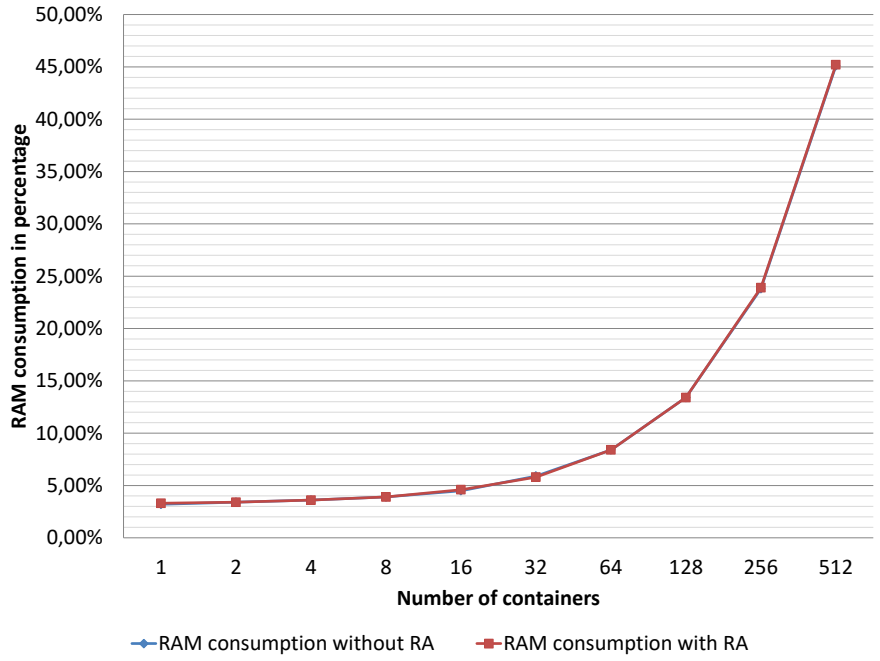

Figure 8.2. RAM consumption on the attester machine as the number of containers increases.

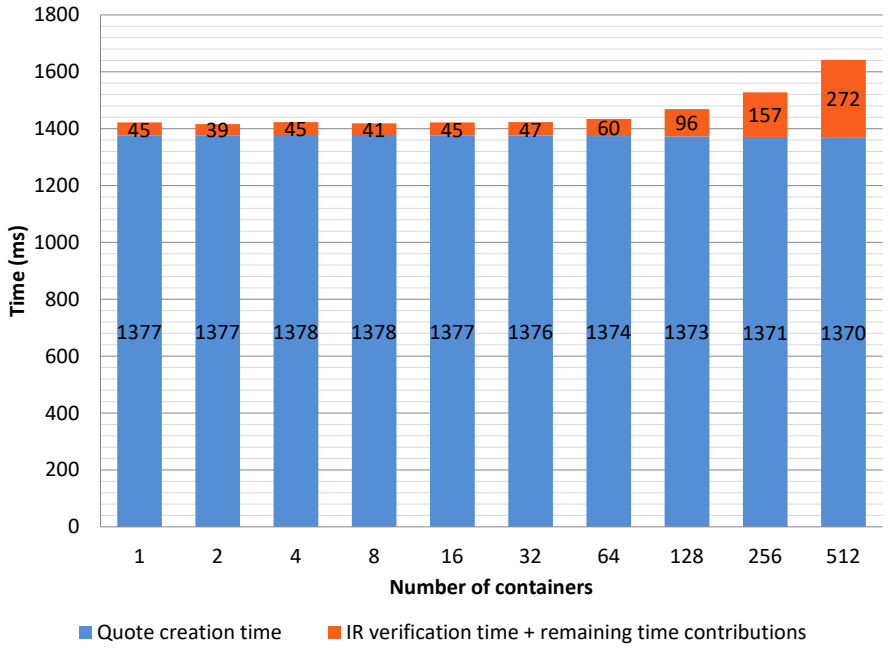

Figure 8.3. Keylime Verifier attestation latency as the number of containers increases.

The last three time contributions depend on the length of the IMA ML file, which in turn depends on the number of containers instantiated on the attesting system; consequently, the optimization described in section 6.4.2, according to which the Verifier does not always ask the Agent for the entire IMA ML but only for the part not yet attested, helps to keep these time contributions low. Observing the figure it can be deduced that:

- the time T_{RA} required for the entire remote attestation process remains constant at 1.4

seconds up to a number of containers equal to 64, and slightly grows up to 1.6 seconds for 512 containers;

- t_{quote}, approximately equal to 1.37 seconds, is the most relevant component of the attestation time and remains constant as the number of containers grows, since it mainly depends on the performance of the hardware TPM and the time required to execute the asymmetric algorithm used for signing the PCRs;
- the time due to $t_{ML\_reading} + t_{IR\_sending} + t_{IR\_verification}$ remains of the order of 10^{-2} seconds up to 32 containers and of the order of 10^{-1} seconds up to 512 containers.

From the data analysis it follows that the proposed solution to perform the integrity verification of a host and the containers running on it is highly scalable, given that the latency of the attestation process remains low and not significantly affected by the increase of the number of containers.

Chapter 9

Conclusions and future work

The specific objective of this thesis was to propose a scalable and efficient solution for periodically monitoring the integrity state of applications deployed in containers, a lightweight virtualization technique widely used in cloud environments and other emerging computational paradigms, such as Fog Computing and Edge Computing. This goal was achieved by developing a new IMA template which, by adding new fields to the entries of the IMA ML, allows a verifier to identify the files accessed by a specific container. The implementation of the solution is based on Keylime, a framework that performs periodic remote attestation of physical platforms and relies on TPM 2.0 as hardware RoT. The Keylime code was appropriately modified in order to make the verifier able to interpret the new template and support attestation of individual containers. Moreover, it was decided to exploit the algorithm agility introduced by TPM 2.0 for the integrity verification of the IMA ML: in this way, it is possible to use an IMA PCR belonging to any of the banks present in the TPM, instead of relying exclusively on the SHA-1 bank, as it happens in the original Keylime implementation. Then, in order to optimize the attestation times, it was decided to check, at each attestation cycle, only the part of the IMA ML not yet attested. Finally, new REST APIs were added to the Keylime components, and others were modified, in order to support the registration of containers associated with a specific Cloud Agent.

It was then decided to integrate the extended Keylime framework in the architecture of TM, a monitoring entity tailored for NFV environments, through the development of a new attestation driver. This was necessary since the attestation frameworks previously used by the TM to attest compute nodes either support TPM 2.0 but not attestation of containers, as Open Cloud Integrity (OpenCIT), or support attestation of containers but not TPM 2.0, as Open Attestation (OAT) extended with the DIVE technology. The work also involved the implementation of a new module, the Whitelists Web Service, wihch is the attempt to unify in a single entity all the logic concerning the creation and management of whitelists, both for physical hosts and containers, allowing the other software components in the TM architecture to access them in a simple way through REST APIs.

The work realized with this thesis provides another mode to perform remote attestation of containers, which takes advantage of interesting properties added by TPM 2.0. As proved by the performance tests carried out in laboratory, the developed solution is highly scalable, adapts to different containerization technologies (e.g., Docker, containerd) and guarantees low attestation latency as the number of deployed containers increases. From the comparison with the examined solutions, it emerges that, for attesting one container, OAT with DIVE takes about 8 seconds, OAT with Container-IMA takes about 2.22 seconds, Keylime with the proposed solution takes about 1.42 seconds. As shown in section 8.3, the attestation time stays below 1.5 seconds up to 128 containers, reaching 1.64 seconds for the attestation of 512 containers. Moreover, it emerges that approximately 93% of the attestation time is spent to create the TPM 2.0 quote; therefore, in order to improve attestation timing, the performance of the TPM 2.0 chip should be improved, or other technologies should be explored to be used as hardware RoT.

The proposed solution is a good starting point for realizing remote attestation of containers with different containerization technologies, but it can be improved in the future by expanding

the objectives to be pursued. The question of ensuring the privacy regarding the measures of containers running in a multi-tenant environment still remains open, so this will be one of the first goals to work on. It would be useful, then, to deepen the characteristics of Kubernetes (K8s), currently the most popular container orchestrator in cloud environments, and also be able to identify, during the remote attestation process, the container's pod, which is the set of one or multiple containers that share the same execution context and represents the smallest unit of computing that can be created and managed in K8s. Moreover, the implementation of the new attestation driver for the TM needs to be refined, as well as the implementation of the Whitelists Web Service in order to get a complete whitelist for a physical host, add support for RPM-based Linux distributions, as currently only Debian-based distributions are supported, and automate whitelist updating when the software of the host system is updated. It would also be useful to deepen the characteristics of the new TEE technologies, such as Intel SGX, ARM TrustZone and AMD SEV, in order to use them as hardware RoT in the RA process. The work carried out in the thesis, therefore, opens the perspective to multiple insights into the numerous scenarios currently emerging in IT infrastructures.

Bibliography

[1] 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

[2] W. Luo, Q. Shen, Y. Xia, and Z. Wu, "Container-ima: A privacy-preserving integrity measurement architecture for containers", 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), Chaoyang District, Beijing, September 23-25, 2019, pp. 487–500

[3] P. S. Tasker, "Trusted computer systems", 1981 IEEE Symposium on Security and Privacy, Oakland (United States, CA), April 1981, DOI 10.1109/sp.1981.10020

[4] Department of Defense, "Department of defense trusted computer system evaluation criteria", The 'Orange Book' Series, pp. 1–129, Palgrave Macmillan UK, 1985, DOI 10.1007/978-1-349-12020-8_1

[5] J. Teo, "Features and benefits of trusted computing", 2009 Information Security Curriculum Development Conference on - InfoSecCD '09, 2009, DOI 10.1145/1940976.1940990

[6] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. V. Doom, "A practical guide to trusted computing", IBM Press, 2007

[7] W. Arthur and D. Challener, "A practical guide to tpm 2.0", Apress open, 2015

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

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

[10] Trusted Computing Group TCG Algorithm Registry, TCG Published, June 25, 2020

[11] I. Pedone, D. Canavese, and A. Lioy, "Trusted computing technology and proposals for resolving cloud computing security problems", Cloud Computing Security: Foundations and Challenges (J. R. Vacca, ed.), pp. 373–386, CRC Press, 2020, DOI 10.1201/9780429055126-31

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

[13] Trusted Computing Group Trusted Platform Module (TPM) 2.0: A Brief Introduction, TCG Published, 2015

[14] IBM's Software TPM 2.0, https://sourceforge.net/projects/ibmswtpm2/

[15] TPM 2.0 Simulator Extraction Script, https://github.com/stwagnr/tpm2simulator

[16] Trusted Computing Group, "Tss overview", TCG TSS 2.0 Overview and Common Structures Specification, pp. 9–12, TCG Published, October 2, 2019

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

[18] Trusted Computing Primary Use Cases, https://trustedcomputinggroup.org/trusted-computing-primary-use-cases/

[19] TPM cryptography cracked, https://web.archive.org/web/20100212050338/https://hackaday.com/2010/02/09/tpm-crytography-cracked/

[20] Schneier on Security, https://www.schneier.com/blog/archives/2015/03/can_the_nsa_bre_1.html

[21] Millions of high-security crypto keys crippled by newly discovered flaw, https://arstechnica.com/information-technology/2017/10/crypto-failure-cripples-millions-of-high-security-keys-750k-estonian-ids/

[22] Information on TPM firmware update for Microsoft Windows systems as announced on Microsoft's patchday on October 10th 2017, https://www.infineon.com/cms/en/product/promopages/tpm-update/

[23] S. Han, W. Shin, J.-H. Park, and H. Kim, "A Bad Dream: Subverting Trusted Platform Module While You Are Sleeping", 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, August 15-17, 2018, pp. 1229–1246

[24] Reinventing the Cold Boot Attack: Modern Laptop Version, https://blog.f-secure.com/podcast-reinventing-cold-boot-attack/

[25] D. Moghimi, B. Sunar, T. Eisenbarth, and N. Heninger, "TPM-FAIL: TPM meets Timing and Lattice Attacks", 29th USENIX Security Symposium (USENIX Security 20) (S. Capkun and F. Roesner, eds.), Boston, MA, August 12-14, 2020, pp. 2057–2073

[26] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a tcg-based integrity measurement architecture", 13th USENIX Security Symposium (USENIX Security 04), San Diego, CA, USA, August 9-13, 2004

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

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

[29] IMA source code, https://elixir.bootlin.com/linux/latest/source/security/integrity/ima

[30] M. Eder, "Hypervisor-vs . container-based virtualization", Proceedings of the Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM) (G. Carle, D. Raumer, and L. Schwaighofer, eds.), Munich, Germany, 2016, pp. 1–7, DOI 10.2313/NET-2016-07-1_01

[31] M. Souppaya, J. Morello, and K. Scarfone, "Application container security guide", NIST Special Publication 800-190, September 2017, DOI 10.6028/NIST.SP.800-190

[32] Container in crescita, ma ancora tante le sfide da affrontare, https://www.cloudtalk.it/container-in-crescita-ma-ancora-tante-le-sfide-da-affrontare/

[33] Linux manual pages: section 7, https://man7.org/linux/man-pages/dir_section_7.html

[34] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: library operating systems for the cloud", SIGPLAN notices, vol. 48, April 2013, pp. 461–472, DOI 10.1145/2499368.2451167

[35] Docker overview, https://docs.docker.com/get-started/overview/

[36] Application Container Security Guide, https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-190.pdf

[37] N. Schear, P. T. Cable, T. M. Moyer, B. Richard, and R. Rudd, "Bootstrapping and maintaining trust in the cloud", Proceedings of the 32nd Annual Conference on Computer Security Applications, New York, NY, USA, December 5 - 8, 2016, pp. 65–77, DOI 10.1145/2991079.2991104

[38] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell, "Linux kernel integrity measurement using contextual inspection", Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing, New York, NY, USA, 2 November 2007, pp. 21–29, DOI 10.1145/1314354.1314362

[39] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks", Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing, New York, NY, USA, 13 November 2009, pp. 49–54, DOI 10.1145/1655108.1655117

[40] Trusted Computing Group, "Attestation key identity certification", Trusted Platform Module Library Part 1: Architecture, pp. 28–29, TCG Published, November 8, 2019

[41] Keylime, https://github.com/keylime/keylime

[42] S. Berger, R. Caceres, K. A. Goldman, R. Perez, R. Sailer, and L. Van Doorn, "vtpm: Virtualizing the trusted platform module", 15th USENIX Security Symposium (USENIX Security 06), Vancouver, B.C. Canada, July 31 - August 4, 2006, pp. 305–320

[43] S. Hosseinzadeh, S. Laurén, and V. Leppänen, "Security in container-based virtualization

through vtpm", 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC), December 2016, pp. 214–219, DOI 10.1145/2996890.3009903

[44] Trusted Computing Group Virtualized Trusted Platform Architecture Specification, TCG Published, September 27, 2011

[45] M. De Benedictis and A. Lioy, "A proposal for trust monitoring in a network functions virtualisation infrastructure", 2019 IEEE Conference on Network Softwarization (NetSoft), Paris (France), June 24-28, 2019, pp. 1–9, DOI 10.1109/NETSOFT.2019.8806655

[46] N. E. I. S. G. (ISG), "Network Functions Virtualisation (NFV); Architectural Framework." ETSI GS NFV 002 v1.1.1, October 2013

[47] N. E. I. S. G. (ISG), "Network Functions Virtualisation (NFV); NFV Security; Problem Statement." ETSI GR NFV 001 v1.2.1, May 2017

[48] N. E. I. S. G. (ISG), "Network Functions Virtualisation (NFV); NFV Security; Security and Trust Guidance ." ETSI GR NFV-SEC 003 v1.2.1, August 2016

[49] N. E. I. S. G. (ISG), "Network Functions Virtualisation (NFV);Use Cases." ETSI GS NFV 002 v1.1.1, October 2013

[50] IMA policy source code, https://elixir.bootlin.com/linux/latest/source/security/integrity/ima/ima_policy.c

Appendix A

User's manual: Testbed

This appendix describes how to configure the environment used for testing the proposed solution.

A.1 Attester machine

In this section we are going to configure a host that will be used as attester in the remote attestation process. The host has to be equipped with a hardware TPM 2.0. The operative system used for testing the proposed solution is Ubuntu Server 20.04 LTS, with a custom Linux kernel containing the IMA module modified as described in appendix C. Install Ubuntu Server 20.04 LTS on a machine (`https://ubuntu.com/download/server`), preferring the manual server installation over other options.

A.1.1 Patching the Linux kernel

First of all, clone the git repository of the stable Linux kernel with the following command, which will create a new directory named `linux-stable` and populate it with the kernel source code:

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
```

The stable repository has several branches starting from `linux-2.6.11.y`; choose the latest stable release branch, which at the time of writing is `linux-5.13.y`:

```
$ cd linux-stable
$ git checkout linux-5.13.y
```

Then, apply the patches containing the modifications to the IMA module described in section 6.3 and provided with the thesis source code:

```
$ git am --signoff < /path/to/patches/0001-ima_cgn_template.patch
$ git am --signoff < /path/to/patches/0002-ima_mns_template.patch
$ git am --signoff < /path/to/patches/0003-entry_hash_256_bit.patch
$ git am --signoff < /path/to/patches/0004-ima_dep_cgn_template.patch
$ git am --signoff < /path/to/patches/0005-ima_cache_clp_patch.patch
```

Now the kernel source code is ready to be compiled.

A.1.2 Compiling and installing the new kernel

Be sure to be in the `linux-stable` directory, then copy the current kernel configuration file into it:

```
$ cp /boot/config-$(uname -r) ./.config
```

Now, in order to customize the kernel configuration file, it is necessary to execute `make menuconfig`, which requires the following libraries to be installed:

```
$ sudo apt-get update
$ sudo apt-get install dist-upgrade make gcc libncurses-dev flex bison
```

Then customize the kernel configuration file:

```
$ make menuconfig
```

Select `Security options --->` in the list that appears on the screen, then scroll down until you reach the `Integrity Measurement Architecture(IMA)` section and select the following configurations:

```
[*] Integrity Measurement Architecture(IMA)
          Default template (ima-dep-cgn) --->
          Default integrity hash algorithm (SHA256) --->
          Default template-hash algorithm (SHA256) --->
[*] IMA cache1 enabled
[*] IMA cache2 enabled
```

Save the modifications and exit. Once this step is complete, the compilation process can start; install the following libraries:

```
$ sudo apt-get install kernel-package libssl-dev
```

Then compile the kernel by running:

```
$ sudo make-kpkg clean
$ sudo fakeroot make-kpkg --initrd --append-to-version=-ima-dep-cgn \
     kernel_image kernel_headers
```

When the kernel compilation is finished, install the new kernel:

```
$ cd ..
$ sudo dpkg -i linux*.deb
```

A.1.3 Booting the kernel

By default, the `grub` bootloader tries to boot the default kernel, which is the newly installed one. If you want to choose the kernel to boot at startup, you need to increase the `GRUB_TIMEOUT` value so that `grub` pauses in the boot menu long enough to choose the kernel. Edit the default grub configuration file `/etc/default/grub`:

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

Uncomment `GRUB_TIMEOUT` and set it to 10 seconds, then comment out `GRUB_TIMEOUT_STYLE`:

```
GRUB_TIMEOUT=10
#GRUB_TIMEOUT_STYLE=hidden
```

Run `update-grub` to update the grub configuration in `/boot`:

```
$ sudo update-grub
```

Now it is time to restart the system; once the boot menu comes up, select the new kernel with suffix "-ima-dep-cgn". Verify that the IMA module is using the `ima-dep-cgn` template and the other configurations you selected through `make menuconfig` by checking the IMA ML file:

```
$ sudo cat /sys/kernel/security/ima/ascii_runtime_measurements
```

You should see a ML file with entries similar to those shown in figure 6.6.

A.1.4 IMA Policy for testbed

An IMA policy [50] specifies which files will be measured by IMA and which components (IMA Measurement, IMA Appraise and IMA Audit) will be involved in the file processing. IMA provides standard policies for measuring and appraising the TCB of the system, extending the *Trusted Boot* and the *Secure Boot* principles to the operating system. It is worth noticing that IMA has some *measurement gaps* that are not closed yet, for example concerning the *extended Berkeley Packet Filter* (eBPF) programs loaded into the kernel-space from user-space, file-like objects such as `stdin` and interpreters. These builtin policies can be configured via the following kernel command line parameters:

- `ima_tcb` (deprecated), that allows to set the ORIGINAL_TCB policy; it was the first builtin IMA policy, available since kernel version 2.6.30, for measuring all critical system files (programs, memory-mapped libraries and files opened by root for reading);
- `ima_policy=`, which can have one or more of the following values, separated by the pipe character"|": `tcb`, that allows to set the DEFAULT_TCB policy; `appraise_tcb`, that enables the local appraising of all files owned by root; `secure_boot`, that allows to appraise firmware, kexec kernel image, kernel moduels and the IMA policy itself; `critical_data`; `fail_securely`;
- `ima_appraise_tcb`, which is equivalent to add "appraise_tcb" to the `ima_policy=` string.

It is also possible to specify a custom IMA policy by writing it inside the `/etc/ima/ima-policy` file, saving the file and rebooting the machine. During the boot, the policy is copied in the `policy` file located in the `securityfs`, typically mounted at `/sys/kernel/security/ima` [11]. Starting from Linux version 4.13, if you do not specify a builtin or a custom IMA policy, the default is "no policy", that is, the accessed files are not processed by IMA.

You can set a custom IMA policy by specifying a set of rules written in a special *IMA Policy language*. Each rule has the format `action [condition ...]`, where:

1. the ***action*** specifies what to do if the file matches the rule, possible actions are:
   - `measure` and `dont_measure`, which specify whether the file measurement has to be added to the IMA ML or not;
   - `appraise` and `dont_appraise`, which specify whether the file measurement has to be compared with a trusted hash value contained in an extended attribute of the file or not;
   - `audit`, which specifies whether the file measurement has to be recorded in the system audit log;
   - `hash` and `dont_hash`, which specify whether the file needs digital signature or not;
2. a ***condition*** specifies a kind of "filter" for the *action*, stating which properties the file should have for matching the rule; conditions can be of two types, *base* and *lsm*, with the possible addition of options; if any of the possible conditions is omitted, it means it is irrilevant for the rule to match, that is, the file may have any value concerning that condition.

The *base* conditions are:

- `func=`, that specifies the way in which the file has been accessed, hence the IMA Hook to be invoked for the file; some of the possible IMA Hooks are:
  - `BPRM_CHECK` for binary programs ("bprm" standing for "BinaryPRograM");
  - `MMAP_CHECK` or the equivalent `FILE_MMAP`, for memory-mapped files;
  - `FILE_CHECK` or the equivalent `PATH_CHECK`, for files opened for reading/writing in the traditional way;

- `mask=` specifies permissions of the accessed file; it can have the following values: `MAY_READ`, `MAY_WRITE`, `MAY_APPEND`, `MAY_EXEC`; if preceded by the character `^`, these values specify the "inverted" mask;
- `fsmagic=` specifies a filesystem format; it can be an hexadecimal number or the corresponding identifier (such as `SYSFS_MAGIC`, `SELINUX_MAGIC`);
- `fsname=` specifies a filesystem name;
- `fsuuid=` specifies a file system UUID;
- `uid=`, `uid>`, `uid<` for identifying the user that accessed the file (`0` identifies the *root* user);
- `euid=`, `euid>`, `euid<` for identifying the effective user that accessed the file (`0` identifies the *root* user);
- `fowner=`, `fowner>`, `fowner<` for specifying the id of the file owner;

The *lsm* conditions leverage file's LSM metadata in order to create more fine graned policies for limiting file measurements only to system sensitive data; for example, not all files opened by root for reading are part of the TCB [28]. The *LSM* specific conditions can be the following six: `subj_user`, `subj_role`, `subj_type`, `obj_user`, `obj_role`, `obj_type`.

The rules are matched in the order in which they are written in the `policy` file, so the more fine grained rules must be specified before the more general ones.

The IMA policy used during testing is the following custom policy:

```
measure func=BPRM_CHECK mask=MAY_EXEC
measure func=FILE_MMAP mask=MAY_EXEC
```

A.1.5 Installing Docker Engine

Now you need to install Docker Engine on the attester machine in order to be able to create and attest Docker containers running on it; for doing that, follow the instructions specified in the reference documentation of the Docker official website https://docs.docker.com/engine/install/ubuntu/.

After having correctly installed Docker Engine, run a container, for example `nginx`:

```
$ sudo docker run -d -p 30000:80 nginx
```

The output of this command is the container full-ID; verify that the IMA module added entries for this container in the `ascii_runtime_measurements` file by running the following command:

```
$ sudo grep "<container full-ID>" /sys/kernel/security/ima/ascii_runtime_measurements
```

If all is correct, you should see a list of entries with the `cgn` field equal to the container full-ID.

A.1.6 Installing Keylime

Keylime requires `libtss2` with version >= 2.4.0, while by default Ubuntu 20.04 has `libtss2` version 2.3.2, so you need to manually build and install `libtss2`, `tpm2-tools` and `tpm2-abrmd`. Before starting, install the following dependencies:

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

TPM 2.0 requirements

1. Manually build and install libtss2 library:

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

2. Manually build and install tpm2-tools:

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

3. Manually build and install TPM 2.0 Resource Manager:

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

Keylime manual installation

Move to the directory containing the version of Keylime provided with the thesis source code, which is Keylime v6.0.0 with the modifications described in section 6.4, and install the python scripts:

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

Copy the Keylime configuration file:

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

Add TPM 2.0 Resource Manager user:

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

Configure TPM Command Transmission Interface (TCTI):

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

Start TPM 2.0 Resource Manager service:

```
$ sudo service tpm2-abrmd start
```

A.1.7 Configuring Keylime Agent

Once you have correctly installed Keylime, open `/etc/keylime.conf`:

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

In the `[general]` section, set the `receive_revocation_ip` parameter to the IP address of the attester machine, for example:

```
receive_revocation_ip = 192.168.1.50
```

The other configuration parameters of the Keylime Agent are located after the `[cloud_agent]` tag. Set `cloudagent_ip` to the IP address of the attester machine, for example:

```
cloudagent_ip = 192.168.1.50
```

Set `registrar_ip` to the IP address of the machine where the Keylime Registrar will run, for example:

```
registrar_ip = 192.168.1.81
```

Set `agent_uuid` to the UUID of the attester machine, for example:

```
agent_uuid = UUID1
```

The tests were performed leaving the default values for the other parameters.

Now, install the Keylime Agent as a systemd service so that it can be managed by using `systemctl`. The directory `services` located in the Keylime root directory contains systemd service files for the Keylime Verifier, the Keylime Agent and the Keylime Registrar, while the script `services/installer.sh` can be used for storing the service files in the systemd path and enabling them at startup. Edit `installer.sh` by commenting out the lines related to the Verifier and the Registrar, as shown below:

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

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

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

Then, move to the Keylime root directory and run the `installer.sh` script for installing the `keylime_agent.service`:

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

Start `keylime_agent.service`:

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

Verify that the status of the `keylime_agent.service` is `active`:

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

A.2 Keylime Verifier and Registrar installation

In this section we are going to configure another machine that will run the Keylime Verifier, Keylime Registrar and Keylime Tenant components. The tests were performed by running these three components on the same machine, although they may be installed and run on different machines. The machine has to be equipped with a TPM 2.0 because these components use the `tpm2-tools` for performing some operations; differently from the attester machine, which has to have a hardware TPM, the TPM 2.0 installed on this machine may also be an emulator as we only need the functionalities provided by the `tpm2-tools`. The operative system installed on the "verifier machine" is Ubuntu Desktop 20.04 LTS (https://ubuntu.com/download/desktop). Install Keylime by following the instructions described in subsection A.1.6, then customize the Keylime configuration file as described in the sections below.

A.2.1 Configuring Keylime Registrar

All configuration parameters concerning Keylime Registrar are located after the `[registrar]` tag in the `/etc/keylime.conf` file. Open the file and set the `registrar_ip` parameter with the IP address of the host, for example:

```
registrar_ip = 192.168.1.81
```

The tests were performed leaving the default values for the other parameters.

A.2.2 Configuring Keylime Verifier

All the configuration parameters concerning Keylime Verifier are located after the `[cloud_verifier]` tag in the `/etc/keylime.conf` file. Open this file and set `cloudverifier_ip` to the host IP address, for example:

```
cloudverifier_ip = 192.168.1.81
```

Set `registrar_ip` to the IP address where the Registrar service will run:

```
registrar_ip = 192.168.1.81
```

Set `revocation_notifier_ip` to the host IP address, for example:

```
revocation_notifier_ip = 192.168.1.81
```

The following two parameters are not part of Keylime v6.0.0, they were added for supporting the proposed solution. Set `containers_dependency` to the name of the software that launches a container; if you installed Docker 20 on the "attester machine", the dependency is `/usr/bin/containerd-shim-runc-`

```
containers_dependency = /usr/bin/containerd-shim-runc-v2
```

Set `containers_dependency_pos` to the position (starting by zero) of the container dependency in the process hierarchy; if you installed Docker 20 on the "attester machine", the dependency position is 2:

```
containers_dependency_pos = 2
```

The tests were performed leaving the other parameters of the `[cloud_verifier]` section to their default values.

A.2.3 Configuring Keylime Tenant

The configuration parameters related to Keylime Tenant are located after the `[tenant]` tag in the `/etc/keylime.conf` file. Open the file and set `cloudverifier_ip` to the IP address of the Verifier:

```
cloudverifier_ip = 192.168.1.81
```

Set `registrar_ip` to the IP address of the Registrar:

```
registrar_ip = 192.168.1.81
```

You can check the Trusted Boot of the "attester machine" via the `tpm_policy` parameter. Read the current values of the PCRs on the "attester machine" by launching the following command on it:

```
$ tpm2_pcrread
```

Copy the values of the Trusted Boot PCRs, with indexes between 0 and 7, of the SHA256 bank in a JSON object with the following format:

```
{
    "0": ["23BD73EC5A35CB441D443DA4E3A234C484845A4E16C1CA3520D8BB8344CE77E3"],
    "1": ["73D3A3D17547D7B057AB3FF522F964466640AD268B45730A8E841134A2966268"],
    "2": ["B3BD342D6060FFE1EBDDD2C7A3D38EDF98CEBC95AC483F9AFEC19A51CA37AB30"],
    ...         ...
    "7": ["4736A834EB08C6C36BA74A2548037438AB9F21D4CAAF6A15CFCE3E3F9F81FD75"]
}
```

Assign this JSON object to the `tpm_policy` parameter, writing it in a single line:

```
tpm_policy = {"0": ["23B..."], "1":["73D..."], "2":["B3B..."], ..., "7":["473..."]}
```

The tests were performed leaving the other parameters of the `[tenant]` section to their default values.

Keylime Tenant checks the validity of the EK certificate sent by the Agent, before sending the encrypted payload and the U share to it. It performs this check by verifying that the EK certificate was issued by a CA whose certificate is stored in the `tpm_cert_store`, a directory containing the trusted issuers' certificates. The location of this directory can be configured via the `tpm_cert_store` parameter, which by default has value `/var/lib/keylime/tpm_cert_store/`. Move to the Keylime root directory and copy the certificates contained in the `tpm_cert_store` directory inside `/var/lib/keylime/tpm_cert_store/`:

```
$ sudo mkdir /var/lib/keylime/tpm_cert_store
$ sudo cp -R ./tpm_cert_store /var/lib/keylime/tpm_cert_store
```

A.2.4 Installing Verifier and Registrar as systemd services

Edit the file `services/installer.sh`, located in the Keylime root directory, and comment out the lines related to the keylime_agent.service, as shown below:

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

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

```
chmod 664 /etc/systemd/system/keylime_verifier.service

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

Install and enable `keylime_registrar.service` and `keylime_verifier.service` by running the `installer.sh` script:

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

Start the services:

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

Verify that they are `active`:

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

A.2.5 Starting remote attestation with Keylime

Now you can verify that the Keylime framework is able to attest a host system and the containers running on it. However, in order to perform runtime attestation, you need to provide the framework with the whitelists related to host system and containers.

Connect to the "attester machine" and launch a couple of containers, for example:

```
$ sudo docker run -d -p 50000:80 nginx
$ sudo docker run -it ubuntu /bin/bash
```

Let us suppose that the "nginx container" has ID `bbe357c4901a` and the "ubuntu container" has ID `019f3a57bcff`. We can create the containers' whitelists by using the IMA ML; obviously, whitelists generated in this way are only useful for debug purposes. Select all the entries related to the "ubuntu container" inside the IMA ML through the following command:

```
$ grep "019f3a57bcff" /sys/kernel/security/ima/ascii_runtime_measurements
```

Create a text file, named for example "allowlist_ubuntu"; then, for each entry of the grep output where the "019f3a57bcff" string is highlighted in the first twelve characters of the `cgn` field, copy the `file hash` and the `file path` fields inside the "allowlist_ubuntu" file with the following format:

```
04a484f27a4b485b28451923605d9b528453d6c098a5a5112bec859fb5f2eea9 /usr/bin/bash
175a52497e658e138f7dce3fbb3ba7d794f302dc7be29f9c9ca79a8cb377554c /usr/bin/groups
e40a8566226636f49d034e6a8acd1b2ffea2c1038eb3d4cf7671a2702f23a139 /usr/bin/dircolors
...
```

Repeat the same procedure for the "nginx container".

Then create a whitelist for the host system by using the program `whitelist_generator.cpp`, provided with the thesis source code. This program generates the whitelist corresponding to a specific path received as parameter. It creates, in the current directory, a file named "whitelist" containing the sha256 hash of all the files found in the path specified as parameter. Compile the program:

```
$ g++ -std=c++17 -L/usr/lib/x86_64-linux-gnu/ \
    -o whitelist_generator whitelist_generator.cpp -lssl -lcrypto
```

In order to create the whitelist corresponding to the path `/usr/bin/`, launch on the "attester machine" the program as follows:

```
$ ./whitelist_generator /usr/bin/
```

Now you need to copy the whitelists of the host and the containers on the "verifier machine". In order to do that, launch on the "verifier machine" the following command:

```
$ nc -l -p 1234 > ./whitelist
```

If you enabled a firewall, ensure that incoming connections on port 1234 are allowed. Supposing that the "verifier machine" has IP address 192.168.1.81, launch on the "attester machine" the command:

```
$ nc -w 3 192.168.1.81 1234 < ./whitelist
```

Repeat the same procedure for the container whitelists, "allowlist_ubuntu" and "allowlist_nginx".

You created for the host system a whitelist related only to the files contained in the path `/usr/bin`, so you need to specify an exclude list in order to exclude all files located in paths other than `/usr/bin` during the remote attestation. On the "verifier machine", create a file named "exclude_host" and write inside it the following regular expression:

```
^(?!/usr/bin/).*$
```

Then create a file, named for example "containers_list", that contains, in each row, the container ID followed by the path to the container-specific whitelist:

```
019f3a57bcff /path/to/allowlist_ubuntu
bbe357c4901a /path/to/allowlist_nginx
```

Finally, create a file that will be sent to the agent after having been ciphered with a random key K_b. For the current test, create an empty file named "payload".

Now you can use the Keylime Tenant CLI, described in more detail in appendix B, for provisioning the verifier with the new agent and starting the remote attestation process:

```
$ sudo keylime_tenant -c add -u UUID1 -t 192.168.1.50 \
     -f payload \
     --allowlist whitelist \
     --exclude exclude_host \
     --cont_list containers_list
```

If all went well, you will see the message "Quote from 192.168.1.50 validated" as last output line: this means that the remote attestation process started. You can check the Agent status via:

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

Instead, if Keylime Tenant does not find the CA issuer certificate in the `tpm_cert_store` directory, you will see the message "TPM Quote from cloud agent is invalid for nonce: <nonce_string>" and the remote attestation process does not start because Keylime Tenant does not trust the TPM of the "attester machine". In this case, verify that the TPM of the "attester machine" has some EK certificates in its NVRAM; the TPM of the machine used for testbeds has two certificates, which can be saved in files by using the following command:

```
$ tpm2_getekcertificate -o RSA_EK_cert.bin -o ECC_EK_cert.bin
```

You need to retrieve the certificates of the CA issuers of the EK certificates and store them in the `tpm_cert_store` directory. Visualize the EK certificates in text format so that you can find where the CA issuer certificates can be downloaded:

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

...
Authority Information Access:
      CA Issuers - URI:http://pki.infineon.com/OptigaRsaMfrCA004/OptigaRsaMfrCA004.crt
...
```

On the "verifier machine", download the CA Issuer certificate and save it in a file:

```
$ curl http://pki.infineon.com/OptigaRsaMfrCA004/OptigaRsaMfrCA004.crt \
     -o OptigaRsaMfrCA004.crt
```

Store the downloaded certificate, in PEM format, inside the `/var/lib/keylime/tpm_cert_store` directory of the "verifier machine":

```
$ openssl x509 -inform der -in ./OptigaRsaMfrCA004.crt \
     -outform pem -out /var/lib/keylime/tpm_cert_store/OptigaRsaMfrCA004.pem
```

Repeat the same procedure for the `ECC_EK_cert.bin` certificate. Now verify that the agent provisioning is successfully completed by launching the following command on the "verifier machine":

```
$ sudo keylime_tenant -c update -u UUID1 -t 192.168.1.50 \
     -f payload \
     --allowlist whitelist \
     --exclude exclude_host \
     --cont_list containers_list
```

Check that the output message is "Quote from 192.168.1.50 validated".

Testing Keylime behaviour

When the remote attestation process starts, if the boot and runtime of the host system are trusted, the Agent will have status "Get Quote" and the attestation is periodically performed; if the continers are trusted, they will have status "Trust". The log files of the Verifier and the Registrar are located at `/var/log/keylime/cloudverifier.log` and `/var/log/keylime/registrar.log`, respectively, on the "verifier machine"; the log file of the Agent is available, on the "attester machine", at `/var/log/keylime/cloudagent.log`. You can check the log files of Verifier and Agent to make sure that the Verifier periodically performs a new attestation of the Agent. Now you can verify that the runtime integrity check is correctly performed for both the containers and the host system. Launch a new command in the "ubuntu container", for example create an empty file:

```
$ root@019f3a57bcff:/# touch prova.txt
```

Now verify that the status of the container "019f3a57bcff" became "Untrust", but the status of the Agent is still "Get Quote" and the periodic attestation is not stopped:

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

You can also verify the correct runtime attestation of the host system by creating a simple script on the "attester machine", saving it in the path `/usr/bin` and launching it: you will see that the status of the Agent will become "Invalid Quote" and the periodic attestation is stopped. Do other tests for verifying the behaviour described in section 6.4.

A.3 Keylime Tenant Webapp installation

In addition to the Tenant CLI, Keylime provides also the Tenant Webapp, a web service that provides the same functionalities of the Tenant CLI through REST APIs, useful when Keylime is used via software tools like the Trust Monitor.

The parameters used by the Tenant Webapp are located in the `[tenant]` and `[webapp]` sections of the `/etc/keylime.conf` file. On the "verifier machine", open this file and configure the parameters of the `[tenant]` section as described in A.2.3. Then, in the `[webapp]` section, set the `webapp_ip` to the IP address of the "verifier machine" and, if you will install the Trust Monitor on the same machine of the Tenant Webapp, set `webapp_port` to a port other than 443; this is needeed because the reverseProxy of the Trust Monitor binds to port 443:

```
webapp_ip = 192.168.1.81
webapp_port = 444
```

Then install the Keylime Tenant Webapp as systemd service; create, in the `/etc/systemd/system` directory, a file named `keylime_webapp.service`:

```
$ sudo nano /etc/systemd/system/keylime_webapp.service
```

Put inside this file the following content:

```
[Unit]
Description=The Keylime Tenant Webapp
After=network.target

[Service]
ExecStart=/usr/local/bin/keylime_webapp

[Install]
WantedBy=default.target
```

Reload the service files to include the new service:

```
$ sudo systemctl daemon-reload
```

Enable the service on every reboot:

```
$ sudo systemctl enable keylime_webapp.service
```

Start the service:

```
$ sudo systemctl start keylime_webapp.service
```

Ensure that the status of the service is `active`:

```
$ sudo systemctl status keylime_webapp.service
```

The log file of the Tenant Webapp is available at `/var/log/keylime/tenant_webapp.log`. The Tenant Webapp provides also a web page at `https://<webapp_ip>:<webapp_port>/webapp/` URI.

Now you can use the Keylime framework via the Tenant Webapp; install Postman on your machine for sending REST requests to it; see appendix D for more information about Tenant Webapp REST APIs. For example, if you previously registered the agent UUID1, you can remove it with a DELETE request at `https://<webapp_ip>:<webapp_port>/agents/UUID1` URI, with Postman or with `curl`:

```
$ curl -X -k DELETE https://192.168.1.81:444/agents/UUID1
```

You can register the agent again by sending a POST request at the same URI; since the agent registration requires to send, in the request body, the host and containers whitelists, you can do it more easily with a script like the following one, by customizing it at your needs:

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
import requests
import sys
import json

if __name__ == '__main__':
    data = {
            "agent_ip": "192.168.1.50",
            "file_data": "",
            "containers": {
                    "019f3a57bcff": {},
                    "bbe357c4901a": {}
```

```
            }
        }
    #read the host whitelist
    with open('whitelist', 'r') as f:
        data["a_list_data"] = f.read().splitlines()

    #read the host exclude list
    with open('exclude_host', 'r') as f:
        data["e_list_data"] = f.read().splitlines()

    #read the ubuntu container whitelist
    with open('allowlist_ubuntu', 'r') as f:
        data["containers"]["019f3a57bcff"]["a_list_data"] = f.read().splitlines()

    #read the nginx container whitelist
    with open('allowlist_nginx', 'r') as f:
        data["containers"]["bbe357c4901a"]["a_list_data"] = f.read().splitlines()

    json_body = json.dumps(data)

    response = requests.post("https://192.168.1.81:444/agents/UUID1", \
        data=json_body, verify=False)

    print("response code: %d" % response.status_code)

    sys.exit(0)
```

Verify that the remote attestation started by consulting the log file of the verifier.

A.4 Whitelists Web Service installation

Move to the directory containing the source code of the Whitelists Web Service and install it on the "verifier machine":

```
$ cd ra-whitelists
$ sudo pip3 install . -r requirements.txt
```

Copy the Whitelists Web Service configuration file in the `/etc/` directory:

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

The Whitelists Web Service configuration parameters are located after the `[web_service]` tag in the `/etc/whitelists.conf` file. Open this file and set the `webservice_ip` parameter to the IP address of the "verifier machine":

```
webservice_ip = 192.168.1.81
```

Set `automatically_update` to `False` so that the server does not populate the database at start up, procedure which may take a long time:

```
automatically_update = False
```

Left the other parameters to their default values and save the file modification.

Copy the `whitelists-packages-sources.list` file in the `/etc/` directory:

```
$ sudo cp whitelists-packages-sources.list /etc/
```

This file contains a list of official repositories from which the Whitelists Web Service will download system packages and has the same syntax as `/etc/apt/sources.list`; by default, it contains the following repositories:

```
deb [arch=amd64,ppc64el,arm64,s390x] https://download.docker.com/linux/ubuntu focal →
    stable
deb [arch=amd64] http://archive.ubuntu.com/ubuntu/ focal main restricted universe →
    multiverse
deb [arch=amd64] http://archive.ubuntu.com/ubuntu/ focal-security main restricted →
    universe multiverse
deb [arch=amd64] http://archive.ubuntu.com/ubuntu/ focal-updates main restricted →
    universe multiverse
deb [arch=amd64] http://archive.ubuntu.com/ubuntu/ focal-backports main restricted →
    universe multiverse
```

You can exclude a repository by adding `#` at the beginning of its line; you can add other repositories in this file if you need, making sure to add the `[arch=...]` option in each line.

Copy the `whitelists-sources-names.txt` file in the `/etc/` directory:

```
$ sudo cp whitelists-sources-names.txt /etc/
```

This file associates the origin and the name of the software to each repository specified in the `whitelists-packages-sources.list`. The default content of this file is the following:

```
http://archive.ubuntu.com/ubuntu/ Ubuntu Operative System
https://download.docker.com/linux/ubuntu Ubuntu Docker
```

If you added new reposetories in the file `whitelists-packages-sources.list`, you have to specify the origin and the software name for each of them in the file `whitelists-sources-names.txt`.

Finally, copy the `whitelists-container-images.txt` file in the `/etc/` directory:

```
$ sudo cp whitelists-container-images.txt /etc/
```

This file specifies a list of Docker images you want to create a whitelist for, with the following information:

1. the method used for retrieving the image, `pull` or `build`;
2. the image-name if the method is `pull`, the path or the URI related to the container source code if the method is `build`;
3. the image-identifier that will be used for referring the image.

By default, this file has the following content:

```
pull ubuntu ubuntu
pull fedora fedora
pull nginx nginx
build https://github.com/keylime/keylime.git#:docker Keylime
```

When the server will start, it will create a whitelist for ubuntu, fedora, nginx and Keylime. You can comment out lines in this file by adding `#` at the beginning of the line, or you can add images to it.

Now install Docker Engine on the "verifier machine" (refer to the official web site https://docs.docker.com/engine/install/). After installation, ensure that Docker Engine is configured to use OverlayFS storage driver (refer to https://docs.docker.com/storage/storagedriver/overlayfs-driver/).

Then install the Whitelists Web Service as systemd service; in the `/etc/systemd/system` directory, create a file named `whitelists_web_service.service`:

```
$ sudo nano /etc/systemd/system/whitelists_web_service.service
```

Put inside it the following content:

```
[Unit]
Description=The Whitelists Web Service
After=network.target

[Service]
ExecStart=/usr/local/bin/whitelists_web_service

[Install]
WantedBy=default.target
```

Now reload the service files to include the new service:

```
$ sudo systemctl daemon-reload
```

Enable the service on every reboot:

```
$ sudo systemctl enable whitelists_web_service.service
```

Start the service:

```
$ sudo systemctl start whitelists_web_service.service
```

Ensure that the status of the service is `active`:

```
$ sudo systemctl status whitelists_web_service.service
```

Now you can send requests to the Whitelists Web Service by using the REST APIs described in appendix E. The log files of the service are located in the `/var/log/whitelists/` directory.

A.4.1 Using Whitelists Web Service with Keylime

Now we are going to use the Whitelists Web Service for creating the whitelists related to host system and containers, in order to use them with Keylime. First of all, you need the list of all packages installed on the "attester machine" in order to create a whitelsit for the host system. For doing that, upgrade the installed packages on the "attester machine":

```
$ sudo apt update
$ sudo apt upgrade
```

Then launch the script `create_list_installed_packages.py`, contained in the root directory of the Whitelists Web Service source code, on the "attester machine"; this script creates a file named `packages_list.txt`, each line of which refers to an installed package and has three fields:

1. the package name;
2. the package version;
3. the architecture;

For example:

```
accountsservice 0.6.55-0ubuntu12~20.04.4 amd64
adduser 3.118ubuntu2 all
alsa-topology-conf 1.2.2-1 all
alsa-ucm-conf 1.2.2-1ubuntu0.8 all
amd64-microcode 3.20191218.1ubuntu1 amd64
...
```

Now you have to create, on the Whitelists Web Service, the resources corresponding to the host whitelist and to the container whitelists, so that Keylime can access them when you provision the new agent. For creating the whitelist of the host UUID1, you have to send a PUT request at the resource `/hosts/UUID1` of the Whitelists Web Service; the body of the PUT request is a JSON object like the following:

```
{
     "architecture": "amd64",
     "hash_algorithms": "sha256",
     "packages_list": "accountsservice 0.6.55-0ubuntu12~20.04.4 amd64\n
                       adduser 3.118ubuntu2 all\n
                       alsa-topology-conf 1.2.2-1 all\n
                       alsa-ucm-conf 1.2.2-1ubuntu0.8 all\n
                       amd64-microcode 3.20191218.1ubuntu1 amd64\n
                       ..."
}
```

where:

- "architecture" corresponds to the architecture of the "attester machine";
- "hash_algorithms" specifies the hash algorithm for the digests contained in the whitelist;
- "packages_list" is a string containing the list of packages installed on the "attester machine", with the format described above.

On the "attester machine" you already created an "ubuntu container" and an "nginx container", and the Whitelists Web Service already created a whitelist for these images; so you only need to create the whitelist resources corresponding to the containers by sending a PUT request at `/containers/<container-ID>` URI. Use the following Python script for creating host and container whitelists, launching it in the same directory where the `packages_list.txt` file is located:

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
import requests
import sys
import json

if __name__ == '__main__':
    data = {
            "architecture": "amd64",
            "hash_algorithms": "sha256"
    }
    with open('packages_list.txt', 'r') as f:
        data["packages_list"] = f.read()

    json_body = json.dumps(data)
    response = requests.put("http://192.168.1.81:8080/hosts/UUID1", data=json_body)
    print("PUT /hosts/UUID1 response code: %d" % response.status_code)

    data = {
            "image_id": "ubuntu",
            "hash_algorithms": "sha256"
    }

    json_body = json.dumps(data)
    response = requests.put("http://192.168.1.81:8080/containers/019f3a57bcff", \
                data=json_body)
    print("PUT /containers/019f3a57bcff response code: %d" % response.status_code)

    data = {
            "image_id": "nginx",
            "hash_algorithms": "sha256"
    }

    json_body = json.dumps(data)
    response = requests.put("http://192.168.1.81:8080/containers/bbe357c4901a", \
                data=json_body)
    print("PUT /containers/bbe357c4901a response code: %d" % response.status_code)
```

```
    sys.exit(0)
```

The Whitelists Web Service takes a while for downloading all the packages specified in the request for creating the host whitelist; when the script finishes, verify that the three whitelists have been created and can be downloaded:

```
$ curl http://192.168.1.81:8080/hosts/UUID1
$ curl http://192.168.1.81:8080/containers/019f3a57bcff
$ curl http://192.168.1.81:8080/containers/bbe357c4901a
```

Now you are ready to use Keylime with the Whitelists Web Service. On the "verifier machine", modify the `containers_list` file by substituting the path to the allowlist with the URI of the container whitelists:

```
019f3a57bcff http://192.168.1.81:8080/containers/019f3a57bcff
bbe357c4901a http://192.168.1.81:8080/containers/bbe357c4901a
```

Finally, launch the following command for updating the agent UUID1 in Keylime:

```
$ sudo keylime_tenant -c update -u UUID1 -t 192.168.1.50 \
     -f payload \
     --allowlist-url http://192.168.1.81:8080/hosts/UUID1
     --cont_list ./containers_list
```

Verify that the attestation is being performed correctly.

A.5 Trust Monitor installation

The TM framework requires Docker Compose to run, so first of all install Docker Compose on the "verifier machine" (refer to the official web site https://docs.docker.com/compose/install/).

When you register an host in the TM with the DriverKeylime, it is necessary that the Keylime Verifier, Registrar, Tenant Webapp and the Whitelists Web Service are running, so you need to install them on the "verifier machine"; follow the instructions described in sections A.2, A.3 and A.4, respectively.

Then, you need to configure a key and a certificate in the `ssl` directory of the `reverseProxy` application. Move to the TM root directory and launch the following command for creating a new key and certificate:

```
$ openssl req -newkey rsa:4096 \
              -x509 \
              -sha256 \
              -days 3650 \
              -nodes \
              -out ./reverseProxy/ssl/certs/test.ra.trust.monitor.chain \
              -keyout ./reverseProxy/ssl/private/test.ra.trust.monitor.key
```

If you create the key and certificate files with names other than `test.ra.trust.monitor.chain` and `test.ra.trust.monitor.key`, respectively, you have to configure them inside the configuration file `./reverseProxy/conf/conf.d/test.ra.trust.monitor.vhost.conf`.

Now, open the file `./trustMonitor/trust_monitor_django/settings.py` and modify the following parameters with the IP address of the Whitelists Web Service and the Keylime Tenant Webapp:

```
WHITELISTS_SERVICE_LOCATION = '192.168.1.81'
KEYLIME_TENANT_LOCATION = '192.168.1.81'
```

Then, from the TM root directory, launch the following command for deploying the TM:

```
$ sudo docker-compose up --build
```

At the end of the build process, run the following command from a different shall, still from the TM root directory, for listing all the containers related to the TM:

```
$ sudo docker-compose ps
```

A.5.1 Using TM with Keylime

If you followed the steps described in the previous sections, you have the agent UUID1 already registered in the Keylime framework so, before registering it through the TM, you have to remove it with the following command:

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

Now, register the agent UUID1 in TM and in Keylime by launching a script like the following one, from a directory containing the `packages_list.txt` file, which is the file containing the list of packages installed on the "attester machine":

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
import requests
import sys
import json

if __name__ == '__main__':
    #Send request to Trust Monitor
    data = {
            "architecture": "amd64",
            "hash_algorithms": "sha256"
        }
    with open('packages_list.txt', 'r') as f:
        data["packages_list"] = f.read()

    distribution = json.dumps(data)

    data = {
            "hostName": "UUID1",
            "address": "192.168.1.50",
            "distribution": distribution,
            "driver": "Keylime",
            "containers": "019f3a57bcff ubuntu\nbbe357c4901a nginx",
            "allowUnknownContainers": 0
        }

    response = requests.post("https://192.168.1.81:443/register_node/", \
            data=data, verify=False)
    print("response code: %d" % response.status_code)

    sys.exit(0)
```

When the agent registration completes, verify the agent status through the TM by sending, with Postman, a POST request at `https://192.168.1.81:443/attest_node/`, with a JSON body like the following one:

```
{
    "node_list": [ { "node": "UUID1" } ]
}
```

Appendix B

User's manual: Keylime Tenant CLI

This appendix focuses on how to use the Keylime Tenant CLI, a utility that allows a user to provision an agent to the Keylime framework. The options described below are those present in Keylime v6.0.0 and those added with this work. Keylime Tenant CLI can be used via the `keylime_tenant` command, which can receive several options. You can instruct `keylime_tenant` about the particular operation you want to perform through `-c`, or `--command`, option which takes one of the following values:

- `-c add` (default value), for registering a new agent in the CV;
- `-c delete`, for removing an agent from the CV;
- `-c update`, for updating an agent in the CV; it is a shortcut for removing and then register again an agent with id <uuid>, so it is equivalent to perform:

  ```
  $ sudo keylime_tenant -c delete -u <uuid>
  $ sudo keylime_tenant -c add -u <uuid>...
  ```

- `-c status`, for retrieving the current status about an agent;
- `-c list`, for retrieving the list of all agents currently registered in the CV;
- `-c reactivate`, for restarting the periodic attestation of an agent which previously failed the integrity check;
- `-c reglist`, for retrieving the list of all agents currently registered in the registrar;
- `-c regdelete`, for removing an agent from the registrar.

The following values are not part of Keylime v6.0.0 and were added for managing containers, allowlists and exclude lists:

- `-c allow_unknown_cont`, for allowing the execution of unregistered containers on an agent;
- `-c forbid_unknown_cont`, for forbidding the execution of unregistered containers on an agent;
- `-c allowlist`, for retrieving or updating the allowlist of hosts and containers;
- `-c exclude`, for retrieving or updating the exclude list of hosts and containers;
- `-c containers`, for managing the containers associated to an agent.

The value assigned to `-c` option determines the list of further parameters that can be passed to the `keylime_tenant` command.

B.1 Adding a new agent to the CV

The command `keylime_tenant -c add` can take the following paramenters:

- `-u <agent_uuid>` or `--uuid <agent_uuid>`, for specifying the UUID of the new agent; if not specified, the default UUID is "D432FBB3-D2F1-4A97-9EF7-75BD81C00000";
- `-v <verifier_ip>` or `--cv <verifier_ip>`, for specifying the IP address of the verifier; if not specified, the default <verifier_ip> is the one assigned to the `cloudverifier_ip` parameter, in the `[tenant]` section of the `/etc/keylime.conf` file;
- `-t <agent_ip>` or `--targethost <agent_ip>`, it is the agent IP address and it is mandatory;
- `-tp <agent_port>` or `--targetport <agent_port>`, it is the port on which the agent's web service is listening; if not specified, the default agent port is the one assigned to the `cloudagent_port` parameter, in the `[cloud_agent]` section of the `/etc/keylime.conf` file;
- `--cv_targethost <agent_ip>`, it is the agent IP address that will be sent to the CV; if not specified, the agent IP address sent to the verifier is the one received through `-t <agent_ip>` option;
- `--allowlist /path/to/host_whitelist`, it is the path to the file that contains the host whitelist; if not specified, the whitelist considered will be the content of the file whose path is assigned to the `ima_allowlist` parameter, in the `[tenant]` section of `/etc/keylime.conf` file; if you do not need to perform runtime integrity check but you want to verify only the trusted boot, assign an empty string to `ima_allowlist` and do not add the `--allowlist` option; a whitelist file has to have the format shown below:

  ```
  hash_file1 /path/to/file1
  hash_file2 /path/to/file2
  ...
  ```

- `--exclude /path/to/host_exclude_list`, it is the path to the file that contains the host exclude list; if not specified, the exclude list will be the content of the file whose path is assigned to the `ima_excludelist` parameter, in the `[tenant]` section of `/etc/keylime.conf` file; if the file does not exist, the host exclude list will be empty; the format of the file is a list of regular expressions that will be used to match the path of the files that do not need to be attested, for example:

  ```
  # enter regex to match file paths to exclude from IMA
  /var/log/wtmp
  /root/etc/fstab
  /boot/grub/grubenv
  /sys/fs/.*
  ```

- `--sign_verification_key ["/path/to/key1", "/path/to/key2", ...]`, it is a list of strings, each of them is the path to a file containing a public key that will be used for the integrity check of the files when the IMA template is "ima-sig"; in this case, the attestation can be performed by using these public keys and the `signature` field of the "ima-sig" template, without considering the whitelist; if not specified, a default empty list is considered;
- `--tpm_policy {"<pcr_idx>": ["<hash1>", "<hash2>", ...], ...}`, it is a JSON object that specifies all the TPM PCRs that you want to check in the quote and the values that you consider trusted for them, that is, it allows to specify a "whitelist" associated to each PCR; if not specified, the default `tpm_policy` associated to the new agent will be the one assigned to the `tpm_policy` parameter, in the `[tenant]` section of the `/etc/keylime.conf` file; in the `tpm_policy` you can specify any PCR except the IMA PCR (by default PCR 10);

- `--vtpm_policy {"<vpcr_idx>": ["<hash1>", "<hash2>", ...], ...}`, it is a JSON object that specifies all the vTPM PCRs that you want to check in the agent's deep-quote and the values that you consider trusted for them; if not specified, the default value is the one assigned to the `vtpm_policy` parameter, in the `[tenant]` section of the `/etc/keylime.conf` file; though it is possible to specify this option, it has no effect since deep-quotes are not supported starting from Keylime v6.0.0;
- `--verify`: this parameter does not need a value, just its presence specifies that you want to check if the agent has correctly derived the bootstrap key K_b, which is the key used to cypher the payload; this check is performed by sending a challenge to the agent, which responds with the $HMAC_{K_b}(challenge)$; if this option is not specified, the key derivation check is not performed;
- `--mb_refstate /path/to/refstate`, it is the path to a file that contains a measured boot reference state policy; if the value is "default", the measured boot reference state policy associated to the new agent will be the file whose path is assigned to the `mb_refstate` parameter, in the `[tenant]` section of the `/etc/keylime.conf` file; this parameter can be specified but is not used in Keylime v6.0.0, it is used starting from Keylime v6.1.0.

The following options concern the payload that the tenant will send to the agent; they are mutually exclusive, so if one of them is specified the others are not admitted, but at least one of them is mandatory:

- `-f /path/to/file` or `--file /path/to/file`, it is the path to a file that contains the payload you want to deliver to the agent; the file content does not need to be ciphered, the tenant will automatically cipher it with a random K_b;
- `-k /path/to/Kb` or `--key /path/to/Kb`, it is the path to a file that contains the bootstrap key K_b together with its splits U and V, with the following format:

  ```
  "Kb in base64"
  "U  in base64"
  "V  in base64"
  ```

  `-k` option requires the presence of the option `-p /path/to/ciphered_payload` or `--payload /path/to/ciphered_payload`, whose value is the path to a file containing the data to be delivered to the agent, ciphered with the K_b specified through the `-k` option;
- `--cert /path/to/ca_dir`, it is the path to the CA directory for the agent certificate and keys; if set to "default", the used directory is `/var/lib/keylime/ca`;
  - if the specified directory does not exist, or it does not contain a file named "cacert.crt", then a new CA directory is created at that path;
  - if the diretory does not contain a file named "<agent_uuid>-private.pem", then a new certificate is created for the agent;
  - if the directory does not contain a file named "RevocationNotifier-private.pem", then a new certificate is created for the revocation notifier; the certificate will be sent to the agent in order to allow it to check the validity of the revocation messages sent by the revocation notifier, while the private key will be sent to the verifier for signing the revocation messages;

  the content of the CA directory is written in a zip file, together with the content of a directory specified through the `--include /path/to/payload_dir` option, which is considered only when the `--cert` option is present; this zip file is then cyphered with a K_b automatically generated by the tenant and delivered to the agent.

Finally, the following options have been added with the thesis work:

- `--cont_list /path/to/file` allows to register containers associated with the new agent; it is the path to a text file, whose format is as follows:

```
019f3a57bcff ./allowlist_019f3a57bcff ./exclude
bbe357c4901a ./allowlist_bbe357c4901a ./exclude_bbe357c4901a
a75524bd319f http://allowlist_server:8080/allowlist_a75524bd319f
...
```

each line can contain three fields:

1. the container ID;
2. the path to the whitelist of the container or the URL from which to download the whitelist;
3. the path to the exclude list of the container (optional);

if this parameter is not specified, the agent will have no associated container;

- `--allow_unknown_containers`, it does not require a value, just its presence specifies that the agent is allowed to run unregistered containers without being considered untrusted by the CV; if not specified, unregistered containers are not admitted for the agent;
- `--allowlist-url <http://remote_server:port/allowlist>`, it is the URL of a remote allowlist; the response body of the HTTP GET performed to that URL has to be in the format of the file specified via `--allowlist`; if this option is present, `--allowlist` can not be specified.

Some agent provisioning examples are listed below.

1. The following command tells keylime to provision a new agent at 192.168.1.50 with id UUID1 and talk to a verifier whose IP is specified in the `/etc/keylime.conf` file. It uses the contents of "allowlist_host" and "exclude_list" files as whitelist and exclude list associated with the host system, respectively. Finally, it encrypts a file named `payload` and sends it to the new agent:

```
$ sudo keylime_tenant -c add -u UUID1 -t 192.168.1.50 \
     -f payload \
     --allowlist allowlist_host \
     --exclude exclude_list
```

2. The following command, as the previous one, tells keylime to provision a new agent at 192.168.1.50 with id UUID1 and talk to a verifier whose IP is specified in the configuration file. It uses the contents of "allowlist_host" and "exclude_host" files as whitelist and exclude list associated with the host system, respectively. It associates with the new agent the list of containers specified in the "containers_list" file where, for each container, the ID, the whitelist and the exclude list are specified. Finally, it encrypts and sends to the agent a .zip file containing:

   - the certificates and the private key contained in `/var/lib/keylime/ca`;
   - all the files contained in the `./payload_dir` directory.

```
$ sudo keylime_tenant -c add -u UUID1 -t 192.168.1.50 \
     --cert default --include ./payload_dir \
     --allowlist allowlist_host \
     --exclude exclude_host \
     --cont_list containers_list
```

3. The following command tells keylime to provision a new agent at 192.168.1.50 with id UUID1 and talk to a verifier at 192.168.1.81. It downloads the host whitelist from the URL "http://192.168.1.81:8080/allowlist_host" and uses the content of the "exclude_host" file as host exclude list. It associates with the new agent the list of containers specified in the "containers_list" file where, for each container, the ID, the whitelist and the exclude list are specified. It admits unregistered containers on the host system. Moreover, it sends to the agent the file `payload` ciphered with the K_b specified in the file `keys`:

```
$ sudo keylime_tenant -c add -u UUID1 -t 192.168.1.50 -v 192.168.1.81 \
    -k keys -p payload \
    --allowlist-url http://192.168.1.81:8080/allowlist_host \
    --exclude exclude_host \
    --cont_list containers_list \
    --allow_unknown_containers
```

B.2 Removing an agent from the CV

The command `keylime_tenant -c delete` can take the following options:

- `-u <agent_uuid>` or `--uuid <agent_uuid>`, it specifies the id of the agent to be cancelled; if not specified, the default UUID "D432FBB3-D2F1-4A97-9EF7-75BD81C00000" is considered;
- `-v <verifier_ip>` or `--cv <verifier_ip>`, it specifies the IP address of the verifier where the agent has to be cancelled; if not specified, the default <verifier_ip> is the one assigned to the `cloudverifier_ip` parameter, in the `[tenant]` section of the `/etc/keylime.conf` file.

Some agent deletion examples are listed below.

1. The following command asks a verifier, whose IP is specified in the `/etc/keylime.conf` file, to stop requesting attestations for the agent UUID1 and remove it from the database:

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

2. The following command asks a verifier at 192.168.1.81 to stop requesting attestations for the agent UUID1 and remove it from the database:

   ```
   $ sudo keylime_tenant -c delete -u UUID1 -v 192.168.1.81
   ```

B.3 Updating an agent in the CV

The command `keylime_tenant -c update` takes the same options as the `keylime_tenant -c add` command, since it internally performs the agent deletion followed by a new registration. See section B.1 for a description of the available options for updating an agent.

B.4 Checking the current status of an agent

The command `keylime_tenant -c status` can take the following options:

- `-u <agent_uuid>` or `--uuid <agent_uuid>`, it specifies the UUID of the agent whose status you want to check; if not specified, the default UUID "D432FBB3-D2F1-4A97-9EF7-75BD81C00000" is considered;
- `-v <verifier_ip>` or `--cv <verifier_ip>`, it specifies the IP address of the verifier where the agent is registered; if not specified, the default <verifier_ip> is the one assigned to the `cloudverifier_ip` parameter, in the `[tenant]` section of the `/etc/keylime.conf` file.

Some examples of checking the status of an agent are listed below.

1. The following command asks a verifier, whose IP is specified in the `/etc/keylime.conf` file, about the current status of the agent UUID1:

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

2. The following command asks a verifier at 192.168.1.81 about the current status of the agent UUID1:

```
$ sudo keylime_tenant -c status -u UUID1 -v 192.168.1.81
```

B.5 Reactivating an agent

The command `keylime_tenant -c reactivate` can take the following options:

- `-u <agent_uuid>` or `--uuid <agent_uuid>`, it specifies the UUID of the agent whose attestation you want to be reactivated; if not specified, the default UUID "D432FBB3-D2F1-4A97-9EF7-75BD81C00000" is considered;
- `-v <verifier_ip>` or `--cv <verifier_ip>`, it specifies the IP address of the verifier where the agent is registered; if not specified, the default <verifier_ip> is the one assigned to the `cloudverifier_ip` parameter, in the `[tenant]` section of the `/etc/keylime.conf` file.

Some examples of reactivating the periodic attestation of an agent are listed below.

1. The following command asks a verifier, whose IP is specified in the `/etc/keylime.conf` file, to reactivate periodic attestation of the agent UUID1:

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

2. The following command asks a verifier at 192.168.1.81 to reactivate periodic attestation of the agent UUID1:

```
$ sudo keylime_tenant -c reactivate -u UUID1 -v 192.168.1.81
```

B.6 Retrieving the list of agents registered in the CV

The command `keylime_tenant -c list` can take the following option:

- `-v <verifier_ip>` or `--cv <verifier_ip>`, it specifies the IP address of the verifier from which you want to receive the list of registered agents; if not specified, the default <verifier_ip> is the one assigned to the `cloudverifier_ip` parameter, in the `[tenant]` section of the `/etc/keylime.conf` file.

Some examples of listing registered agents are shown below.

1. The following command asks the list of all agents registered in a verifier whose IP is specified in the `/etc/keylime.conf` file:

```
$ sudo keylime_tenant -c list
```

2. The following command asks the list of all agents registered in a verifier at 192.168.1.81:

```
$ sudo keylime_tenant -c list -v 192.168.1.81
```

B.7 Retrieving the list of agents registered in the registrar

The command `keylime_tenant -c reglist` does not take additional options; it asks a registrar, whose IP address and port are configured in the `registrar_ip` and `registrar_port` parameters of the `/etc/keylime.conf` file, in the `[Tenant]` section, to list all the agents currently registered in it:

```
$ sudo keylime_tenant -c reglist
```

B.8 Removing an agent from the registrar

The command `keylime_tenant -c regdelete` can take the following option:

- `-u <agent_uuid>` or `--uuid <agent_uuid>`, it specifies the UUID of the agent you want to remove from the registrar; if not specified, the default UUID "D432FBB3-D2F1-4A97-9EF7-75BD81C00000" is considered.

The following command asks a registrar, whose IP address and port are configured in the `registrar_ip` and `registrar_port` parameters of the `/etc/keylime.conf` file, in the `[Tenant]` section, to remove the agent UUID1:

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

B.9 Allowing/forbidding unknown containers in the agent

The command `keylime_tenant -c allow_unknown_cont` and its dual `-c forbid_unknown_cont` can take the following options:

- `-u <agent_uuid>` or `--uuid <agent_uuid>`, it specifies the UUID of the agent for which you want to turn on/off the option to run unregistered containers; if not specified, the default UUID "D432FBB3-D2F1-4A97-9EF7-75BD81C00000" is considered;
- `-v <verifier_ip>` or `--cv <verifier_ip>`, it specifies the IP address of the verifier where the agent is registered; if not specified, the default <verifier_ip> is the one assigned to the `cloudverifier_ip` parameter, in the `[tenant]` section of the `/etc/keylime.conf` file.

Some examples of turning on/off the flag for running unknown containers are listed below.

1. The following command turns on the flag for running unknown containers on an agent UUID1, registered in a verifier whose IP is specified in the `/etc/keylime.conf` file:

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

2. The following command turns off the flag for running unknown containers on an agent UUID1, registered in a verifier at 192.168.1.81:

   ```
   $ sudo keylime_tenant -c forbid_unknown_cont -u UUID1 -v 192.168.1.81
   ```

B.10 Managing containers

The command `keylime_tenant -c containers` can take the following options:

- `-u <agent_uuid>` or `--uuid <agent_uuid>`, it specifies the UUID of the agent whose containers you want to manage; if not specified, the default UUID "D432FBB3-D2F1-4A97-9EF7-75BD81C00000" is considered;
- `-v <verifier_ip>` or `--cv <verifier_ip>`, it specifies the IP address of the verifier where the agent is registered; if not specified, the default <verifier_ip> is the one assigned to the `cloudverifier_ip` parameter, in the `[tenant]` section of the `/etc/keylime.conf` file;
- `--cont_id <container_id>`, it specifies the ID of the container you want to manage; if this option is specified, the options `--cont_list` and `--replace_cont_list` are not admitted;
- `--allowlist /path/to/cont_allowlist`, it is the path to a file containing the allowlist of the container whose ID is specified through `--cont_id`; this option requires the `--cont_id` option and is not admitted with `--cont_list`;
- `--allowlist-url <http://remote_server:port/allowlist>`, it is the URL of a remote allowlist for the container whose ID is specified via `--cont_id`; this option requires the `--cont_id` option and, if it is present, `--allowlist` can not be specified;
- `--exclude /path/to/cont_exclude_list`, it is the path to a file containing the exclude list of the container whose ID is specified via `--cont_id`; this option requires the `--cont_id` option and is not admitted with `--cont_list`;
- `--cont_list /path/to/containers_list`, it is the path to a file containing the list of containers you want to add/replace in the agent; the format of this file was described in section B.1;
- `--replace_cont_list`, it can be used with the option `--cont_list` and does not receive a value, its presence implies the replacement of the list of containers currently registered in the agent with the one provided through `--cont_list`.

Some examples of containers management are listed below.

1. The following command asks a verifier at 192.168.1.81 the list of all containers registered in the agent UUID1:

```
$ sudo keylime_tenant -c containers -u UUID1 -v 192.168.1.81
```

2. The following command asks a verifier at 192.168.1.81 the information (operational state, allowlist, exclude list) related to the container "019f3a57bcff", registered in the agent UUID1:

```
$ sudo keylime_tenant -c containers -u UUID1 -v 192.168.1.81 \
   --cont_id 019f3a57bcff
```

3. The following command asks a verifier at 192.168.1.81 to substitute the list of containers currently registered in the agent UUID1 with the one specified with `--cont_list`:

```
$ sudo keylime_tenant -c containers -u UUID1 -v 192.168.1.81 \
   --cont_list containers_list --replace_cont_list
```

4. The following command asks a verifier at 192.168.1.81 to add the containers specified via `--cont_list` to those already registered in the agent UUID1:

```
$ sudo keylime_tenant -c containers -u UUID1 -v 192.168.1.81 \
   --cont_list containers_list
```

5. The following command asks a verifier, whose IP is specified in the `/etc/keylime.conf` file, to update the whitelist and exclude list associated of the container "019f3a57bcff", registered in the agent UUID1:

```
$ sudo keylime_tenant -c containers -u UUID1 \
   --cont_id 019f3a57bcff \
   --allowlist allowlist_019f3a57bcff \
   --exclude exclude_019f3a57bcff
```

B.11 Managing the allowlist

The command `keylime_tenant -c allowlist` can take the following options:

- `-u <agent_uuid>` or `--uuid <agent_uuid>`, it specifies the UUID of the agent; if not specified, the default UUID "D432FBB3-D2F1-4A97-9EF7-75BD81C00000" is considered;
- `-v <verifier_ip>` or `--cv <verifier_ip>`, it specifies the IP address of the verifier where the agent is registered; if not specified, the default <verifier_ip> is the one assigned to the `cloudverifier_ip` parameter, in the `[tenant]` section of the `/etc/keylime.conf` file;
- `--allowlist /path/to/allowlist`, it is the path to the file containing the allowlist that will substitute the current one;
- `--allowlist-url <http://remote_server:port/allowlist>`, it is the URL of a remote allowlist that will substitute the current one; if this option is present, `--allowlist` can not be specified;
- `--patch /path/to/patch_file`, it is the path to a file that specifies how the allowlist has to be patched; the format of the file is as follows:

```
#==================================
[delete]
#==================================
/path/to/file_to_be_removed_1
/path/to/file_to_be_removed_2
...

#==================================
[put]
#==================================
<hash_1> /path/to/file_1
<hash_2> /path/to/file_2
...
```

  - after the `[delete]` tag you have to list the paths of the files you want to remove from the allowlist;
  - after the `[put]` tag you have to list the hash followed by the path of the files you want to add or update in the allowlist;

- `--cont_id <container_id>`, it is the ID of the container whose allowlist you want to manage.

Some examples of allowlist management are listed below.

1. The following command asks a verifier at 192.168.1.81 the host allowlist associated with the agent UUID1:

```
$ sudo keylime_tenant -c allowlist -u UUID1 -v 192.168.1.81
```

2. The following command asks a verifier at 192.168.1.81 to update the host allowlist associated with the agent UUID1, replacing the current one with the one contained in the file "allowlist_host":

```
$ sudo keylime_tenant -c allowlist -u UUID1 -v 192.168.1.81 \
   --allowlist allowlist_host
```

3. The following command asks a verifier at 192.168.1.81 to update the host allowlist associated with the agent UUID1, adding, updating or removing some entries as specified in the file "patch_file":

```
$ sudo keylime_tenant -c allowlist -u UUID1 -v 192.168.1.81 --patch patch_file
```

4. The following command asks a verifier at 192.168.1.81 the allowlist of the container with ID "019f3a57bcff", associated with the agent UUID1:

```
$ sudo keylime_tenant -c allowlist -u UUID1 -v 192.168.1.81 \
   --cont_id 019f3a57bcff
```

5. The following command asks a verifier at 192.168.1.81 to update the allowlist of the container "019f3a57bcff" associated with the agent UUID1, replacing the current one with the one downloaded from "http://192.168.1.81:8080/containers/019f3a57bcff":

```
$ sudo keylime_tenant -c allowlist -u UUID1 -v 192.168.1.81 \
   --cont_id 019f3a57bcff \
   --allowlist-url http://192.168.1.81:8080/containers/019f3a57bcff
```

6. The following command asks a verifier, whose IP is specified in the `/etc/keylime.conf` file, to update the allowlist of a container with ID "019f3a57bcff" associated with the agent UUID1, adding, updating or removing some entries as specified in the file "patch_file":

```
$ sudo keylime_tenant -c allowlist -u UUID1 --cont_id 019f3a57bcff \
   --patch patch_file
```

B.12 Managing the exclude list

The command `keylime_tenant -c exclude` can take the following options:

- `-u <agent_uuid>` or `--uuid <agent_uuid>`, it specifies the UUID of the agent; if not specified, the default UUID "D432FBB3-D2F1-4A97-9EF7-75BD81C00000" is considered;
- `-v <verifier_ip>` or `--cv <verifier_ip>`, it specifies the IP address of the verifier where the agent is registered; if not specified, the default <verifier_ip> is the one assigned to the `cloudverifier_ip` parameter, in the `[tenant]` section of the `/etc/keylime.conf` file;
- `--exclude /path/to/exclude_list`, it is the path to the file containing the exclude list that will substitute the current one;
- `--cont_id <container_id>`, it is the ID of the container whose exclude list you want to manage.

Some examples of exclude list management are listed below.

1. The following command asks a verifier at 192.168.1.81 the host exclude list associated with the agent UUID1:

```
$ sudo keylime_tenant -c exclude -u UUID1 -v 192.168.1.81
```

2. The following command asks a verifier at 192.168.1.81 to update the host exclude list associated with the agent UUID1, replacing the current one with the one contained in the file "exclude_host":

```
$ sudo keylime_tenant -c exclude -u UUID1 -v 192.168.1.81 \
   --exclude exclude_host
```

3. The following command asks a verifier, whose IP is specified in the `/etc/keylime.conf` file, the exclude list of the container with ID "019f3a57bcff", associated with the agent UUID1:

```
$ sudo keylime_tenant -c exclude -u UUID1 --cont_id 019f3a57bcff
```

4. The following command asks a verifier at 192.168.1.81 to update the exclude list of the container "019f3a57bcff" associated with the agent UUID1, replacing the current one with the one contained in the file "exclude_019f3a57bcff":

```
$ sudo keylime_tenant -c exclude -u UUID1 -v 192.168.1.81 \
   --cont_id 019f3a57bcff --exclude exclude_019f3a57bcff
```

Appendix C

Programmer's manual: IMA patches

This Appendix describes the implementation of the IMA patches created for the thesis work.

C.1 IMA Template Framework

The source code of the Linux kernel can be browsed at https://elixir.bootlin.com/linux/latest/source. All the source files concerning the IMA module are located in the directory /security/integrity/ima and, inside it, the files pertaining to the IMA template management mechanism are the following three:

1. ima_template.c containing the definitions of the functions used to initialize the IMA template that has been selected;
2. ima_template_lib.c containing the definitions of the functions used to initialize the template fields and to write them in the IMA Measurement Log (ML) files in binary and ASCII formats;
3. ima_template_lib.h containing the declarations of the prototypes of the functions defined in `ima_template_lib.c`.

IMA supports some builtin templates, defined in the source file ima_template.c (listing C.1).

Listing C.1. Extract of ima_template.c (lines 19-26)

```
static struct ima_template_desc builtin_templates[] = {
    {.name = IMA_TEMPLATE_IMA_NAME, .fmt = IMA_TEMPLATE_IMA_FMT},
    {.name = "ima-ng", .fmt = "d-ng|n-ng"},
    {.name = "ima-sig", .fmt = "d-ng|n-ng|sig"},
    {.name = "ima-buf", .fmt = "d-ng|n-ng|buf"},
    {.name = "ima-modsig", .fmt = "d-ng|n-ng|sig|d-modsig|modsig"},
    {.name = "", .fmt = ""},    /* placeholder for a custom format */
};
```

It is possible to specify the builtin template that the IMA module will use via the kernel boot parameter `ima_template`, giving it the name of the chosen template (i.e., `ima_template=ima-sig`). If this parameter is not specified, by default IMA uses `ima-ng`. Each template is defined by its format string `fmt`, which specifies the identifiers of the fields that a template entry will contain. The fields supported by IMA are defined in the same source file (listing C.2).

Listing C.2. Extract of ima_template.c (lines 31-48)

```
static const struct ima_template_field supported_fields[] = {
    {.field_id = "d", .field_init = ima_eventdigest_init,
     .field_show = ima_show_template_digest},
    {.field_id = "n", .field_init = ima_eventname_init,
     .field_show = ima_show_template_string},
    {.field_id = "d-ng", .field_init = ima_eventdigest_ng_init,
     .field_show = ima_show_template_digest_ng},
    {.field_id = "n-ng", .field_init = ima_eventname_ng_init,
     .field_show = ima_show_template_string},
    {.field_id = "sig", .field_init = ima_eventsig_init,
     .field_show = ima_show_template_sig},
    {.field_id = "buf", .field_init = ima_eventbuf_init,
     .field_show = ima_show_template_buf},
    {.field_id = "d-modsig", .field_init = ima_eventdigest_modsig_init,
     .field_show = ima_show_template_digest_ng},
    {.field_id = "modsig", .field_init = ima_eventmodsig_init,
     .field_show = ima_show_template_sig},
};
```

- `d`: the digest of the event, computed with SHA-1 or MD5 algorithm (typically the digest of a file's contents);
- `n`: the name of the event with size up to 255 characters (typically the pathname of a file);
- `d-ng`: the digest of the event, computed with an arbitrary hash algorithm; the field format is "<hash_algo>:digest", where the digest prefix is a string that specifies the used hash algorithm, defined through the `ima_hash` kernel boot parameter;
- `n-ng`: the name of the event without size limitations;
- `sig`: the file signature, if present;
- `buf`: the buffer data used to generate the digest without size limitations;
- `d-modsig`: the digest of the event, computed without the appended file signature;
- `modsig`: the appended file signature.

If we want ML entries with a format other than the one defined by builtin templates, we can do this through the kernel boot parameter `ima_template_fmt=`, giving it the desired format string, with the constraint that the specified fields are among those supported by IMA. For example, we could define a template format as `d-ng|n-ng|buf|sig` and specify it to the kernel via the boot parameter `ima_template_fmt=d-ng|n-ng|buf|sig`. Instead, if we want the formatting string to contain a field that IMA does not support, it is necessary to modify the kernel for defining the desired field.

C.1.1 `ima-cgn` template implementation

As described in section 6.3.1, the `ima-cgn` template has a field containing the control group name of the process that generated the current ME; this kind of field is not among those supported by IMA, so we added it to the `supported_fields` array:

```
static const struct ima_template_field supported_fields[] = {
    ...
    {.field_id = "cgn", .field_init = ima_eventcgn_init,
     .field_show = ima_show_template_string},
};
```

where:

1. `"cgn"` is the field identifier that has to be inserted in the template format string;
2. `ima_eventcgn_init` is the function that initializes the field value for a given ME and will have to be properly defined;
3. `ima_show_template_string` is the function that writes the field value in the MLs, already defined in the IMA module for `"n"` and `"n-ng"` fields (ima_template_lib.c, lines 147-151).

We defined a new builtin template with name `"ima-cgn"` and format string `"cgn|d-ng|n-ng"`, where "d-ng" is the file digest and "n-ng" is the file path:

```
static struct ima_template_desc builtin_templates[] = {
    ...
    {.name = "ima-cgn", .fmt = "cgn|d-ng|n-ng"},
};
```

We added the prototype of the `ima_eventcgn_init` function in the `ima_template_lib.h` file:

```
int ima_eventcgn_init(struct ima_event_data *event_data,
  struct ima_field_data *field_data);
```

and its definition in the `ima_template_lib.c` file (listing C.3).

Listing C.3. ima_template_lib.c

```
1  int ima_eventcgn_init(struct ima_event_data *event_data,
2   struct ima_field_data *field_data)
3  {
4      char *cgroup_name_str = NULL;
5      struct cgroup *cgroup = NULL;
6      int rc = 0;
7
8      cgroup_name_str = kmalloc(NAME_MAX, GFP_KERNEL);
9      if (!cgroup_name_str)
10         return -ENOMEM;
11
12     cgroup = task_cgroup(current, 1);
13     if (!cgroup)
14         goto out;
15     rc = cgroup_name(cgroup, cgroup_name_str, NAME_MAX);
16     if (!rc)
17         goto out;
18
19     rc = ima_write_template_field_data(cgroup_name_str, strlen(cgroup_name_str), ↪
       DATA_FMT_STRING, field_data);
20
21     kfree(cgroup_name_str);
22
23     return rc;
24
25 out:
26     return ima_write_template_field_data("-", 1, DATA_FMT_STRING, field_data);
27 }
```

`ima_eventcgn_init` defines and initializes three local variables: `cgroup_name_str` (line 4) will contain the name of the control group, `cgroup` (line 5) will refer to data structure that describes the selected cgroup, `rc` (line 6) will contain the return code of function calls. It invokes `kmalloc()` to allocate the memory needed to contain the cgroup name (line 8), taking as parameters:

- `NAME_MAX`, which specifies the number of bytes to be allocated; it is a `#define` contained in the header file /include/uapi/linux/limits.h that corresponds to the number 255;
- `GFP_KERNEL`, typically used for kernel-internal allocations.

After verifying that the allocation was successful, it invokes `task_cgroup(current, 1)` (line 12) in order to retrieve the control group corresponding to hierarchy ID = 1 (the `name=systemd` controller is associated to this hierarchy, as shown in figure 6.2) for the current task, which can be accessed through `current`, defined in the file /include/asm-generic/current.h. `current` refers to a data structure of type `struct task_struct` (/include/linux/sched.h) containing all information regarding a Linux task. Then, it invokes `cgroup_name()` (line 15), which puts the name of the `cgroup` previously retrieved in the `cgroup_name_str` buffer. Then it invokes `ima_write_template_field_data()` (line 19) for writing the template value in the `field_data` parameter, specifying that the data format is `DATA_FMT_STRING`. The ima_template_lib.c file contains the definition of `ima_write_template_field_data()`.

We added the `ima-cgn` template in the Kconfig file of the IMA module, so that it can be configured as the default IMA template through the `make menuconfig` command before compiling the kernel (listing C.4).

Listing C.4. Extract of Kconfig (lines 62-91)

```
choice
        prompt "Default template"
        default IMA_NG_TEMPLATE
        depends on IMA
        help
          Select the default IMA measurement template.
        ...

        config IMA_CGN_TEMPLATE
               bool "ima-cgn"
endchoice

config IMA_DEFAULT_TEMPLATE
        string
        depends on IMA
        ...
        default "ima-cgn" if IMA_CGN_TEMPLATE
```

C.1.2 `ima-dep-cgn` template implementation

This section describes the code for the `ima-dep-cgn` template, presented in section 6.3.2. This template defines a new field, named `"dep"`, containing the dependencies of the process that generated the ME; it was added in the `supported_fields` array defined in the `ima_template.c` source file (listing C.2):

```
static const struct ima_template_field supported_fields[] = {
    ...
    {.field_id = "dep", .field_init = ima_eventdep_init,
     .field_show = ima_show_template_string},
};
```

The `"dep"` field has `ima_eventdep_init` as initialization function, whose code is presented below, and `ima_show_template_string` as function for writing the field in the MLs, used also for the fields `"n"`, `"n-ng"` and `"cgn"`. A new template was added to the builtin templates (listing C.1), with name `"ima-dep-cgn"` and format string `"dep|cgn|d-ng|n-ng"`:

```
static struct ima_template_desc builtin_templates[] = {
    ...
    {.name = "ima-dep-cgn", .fmt = "dep|cgn|d-ng|n-ng"},
};
```

We added the prototype of the `ima_eventdep_init` initialization function of the `"dep"` field in the `ima_template_lib.h` header file:

```
int ima_eventdep_init(struct ima_event_data *event_data,
  struct ima_field_data *field_data);
```

and its definition in the `ima_template_lib.c` source file (listing C.5).

Listing C.5. ima_template_lib.c

```
1  int ima_eventdep_init(struct ima_event_data *event_data,
2   struct ima_field_data *field_data)
3  {
4      int count = 0, rc;
5      char *paths_buf = NULL, *pathbuf = NULL;
6      const char *pathname = NULL;
7      char filename[NAME_MAX];
8      struct task_struct *curr_task = NULL;
9      struct file *exe_file = NULL;
10     char comm[TASK_COMM_LEN];
11
12     //get number of ancestors for current task
13     for (curr_task = current; curr_task && curr_task->pid; curr_task = →
       curr_task->real_parent)
14         count++;
15
16     if (curr_task)
17         count++;
18
19     paths_buf = kmalloc(PATH_MAX*count+count-1, GFP_KERNEL);
20     if (!paths_buf)
21         return -ENOMEM;
22
23     paths_buf[0] = '\0';
24     for (curr_task = current; curr_task && curr_task->pid; curr_task = →
       curr_task->real_parent) {
25         exe_file = get_task_exe_file(curr_task);
26         if (!exe_file) {
27             get_task_comm(comm, curr_task);
28             strcat(paths_buf, comm);
29             strcat(paths_buf, ":");
30             continue;
31         }
32
33         pathname = ima_d_path(&exe_file->f_path, &pathbuf, filename);
34
35         strcat(paths_buf, pathname);
36         strcat(paths_buf, ":");
37     }
38     if (curr_task) {
39         exe_file = get_task_exe_file(curr_task);
40         if(!exe_file) {
41             get_task_comm(comm, curr_task);
42             strcat(paths_buf, comm);
43         } else {
44             pathname = ima_d_path(&exe_file->f_path, &pathbuf, filename);
45             strcat(paths_buf, pathname);
46         }
47     }
48
49     rc = ima_write_template_field_data(paths_buf, strlen(paths_buf), →
       DATA_FMT_STRING, field_data);
50
51     kfree(paths_buf);
52
53     return rc;
```

```
54 }
```

The `ima_eventdep_init()` function has the purpose to create a string containing the paths, colon separated, of the executables of the process that generated the current ME and of all its ancestors and to record this string in the template field. In order to do this, firstly it determines the number of ancestors, stored in the `count` variable, of the current process (lines 13-14), starting from `current` (that refers to a `struct task_struct` describing the current process) and retrieving the ancestors via the `real_parent` field of the `struct task_struct` (/include/linux/sched.h, line 863); in order to include also the `init` process, `count` is incremented by 1 out of the cycle (lines 16-17). The control statement of the `for` cycle checks both the pointer and the PID since normally `real_parent` never becomes `NULL`, so PID=0 (which corresponds to the `init` process) is used to break the cycle. The function, after having allocated the memory (line 19), creates the dependencies list (lines 24-47). The size of the allocated memory has to consider the number of processes in the hierarchy (`count`), `PATH_MAX` (defined in include/uapi/linux/limits.h with value 4096 and used as the max length of Linux path) and `count-1` separator characters. The `for` cycle (lines 24-37) walks through the ancestors for getting the executable file (line 25). If `exe_file` is `NULL` (as for kernel tasks), the executable name is substituted with the command name contained in the task (the `comm` field of the `struct task_struct`, /include/linux/sched.h line 960), retrieved through the `get_task_comm()` function (line 27), defined in /include/linux/sched.h; otherwise the absolute path of the executable is retrieved by invoking the `ima_d_path()` function (line 33), defined in /security/integrity/ima/ima_api.c; the path is concatenated with the other paths in the `paths_buf` variable (lines 35-36). The last task in the chain of ancestors is the `init` task, which is managed out of the cycle (lines 38-47). The function stores the created dependencies list in the `field_data` parameter by invoking `ima_write_template_field_data()` (line 49), then it releases the buffer previously allocated for the string (line 51) and returns the control to the caller (line 53).

As for `ima-cgn` template, we added `ima-dep-cgn` template to the IMA Kconfig file so that it can be choosen as the default template via the `make menuconfig` command, before compiling the kernel:

```
choice
        ...
        config IMA_DEP_CGN_TEMPLATE
                bool "ima-dep-cgn"
endchoice

config IMA_DEFAULT_TEMPLATE
        ...
        default "ima-dep-cgn" if IMA_DEP_CGN_TEMPLATE
```

C.1.3 `ima_template_hash=` kernel boot parameter implementation

This section presents the code added to the IMA module in order to allow a user to choose the template-hash written in the MLs. To this purpose, we defined a new kernel boot parameter, named `ima_template_hash=`, to which the user assigns the desired hash algorithm for the template-hash. We added the function `template_hash_setup()` (listing C.6) to the `ima_main.c` source file for initializing, on the basis of the boot parameter value, the global variable `ima_template_hash_algo` that contains the identifier of the selected hash algorithm.

Listing C.6. ima_main.c

```
1 int ima_template_hash_algo = HASH_ALGO_SHA1;
2 static int template_hash_setup_done;
3 ...
4 static int __init template_hash_setup(char *str)
5 {
6     int i;
7
```

```
8       if (template_hash_setup_done)
9           return 1;
10
11      i = match_string(hash_algo_name, HASH_ALGO__LAST, str);
12      if (i < 0) {
13          pr_err("invalid template-hash algorithm \"%s\"", str);
14          return 1;
15      }
16
17      ima_template_hash_algo = i;
18
19      template_hash_setup_done = 1;
20      return 1;
21  }
22  __setup("ima_template_hash=", template_hash_setup);
```

We added the invocation to the `template_hash_setup()` in the `init_ima()` function, defined in ima_main.c file, in order to initialize the `ima_template_hash_algo` variable with the default value specified in the IMA `Kconfig` file, in the case the user has not set the kernel boot parameter `ima_template_hash=`.

Listing C.7. Extract from ima_main.c

```
982 static int __init init_ima(void)
983 {
984     ...
985     template_hash_setup(CONFIG_IMA_DEFAULT_TEMPLATE_HASH);
986     ...
987     if (error && strcmp(hash_algo_name[ima_template_hash_algo], ↪
        CONFIG_IMA_DEFAULT_TEMPLATE_HASH) != 0) {
988         pr_info("Allocating %s failed, going to use default template-hash algorithm ↪
        %s \n", hash_algo_name[ima_template_hash_algo], ↪
        CONFIG\_IMA\_DEFAULT\_TEMPLATE\_HASH);
989         template_hash_setup_done = 0;
990         template_hash_setup(CONFIG_IMA_DEFAULT_TEMPLATE_HASH);
991         error = ima_init();
992     }
993     ...
994 }
```

In the file `ima_crypto.c`, we defined another global variable, `ima_template_hash_algo_idx`, which specifies the index of the template-hash calculated with the selected hash algorithm inside the array containing all the template-hashes computed for a given ML entry. We added some code in the `ima_init_crypto()` function, defined in the ima_crypto.c file, in order to initialize `ima_template_hash_algo_idx` (listing C.8).

Listing C.8. Extract from ima_crypto.c

```
65  int ima_template_hash_algo_idx __ro_after_init;
66  ...

115 int __init ima_init_crypto(void)
116 {
117     ...
118     ima_template_hash_algo_idx = -1;
119
120     for (i = 0; i < NR_BANKS(ima_tpm_chip); i++) {
121         algo = ima_tpm_chip->allocated_banks[i].crypto_id;
122         ...
123         if (algo == ima_template_hash_algo)
124             ima_template_hash_algo_idx = i;
```

```
125     }
126
127     if (ima_sha1_idx < 0) {
128         ...
129         if (ima_template_hash_algo == HASH_ALGO_SHA1)
130             ima_template_hash_algo_idx = ima_sha1_idx;
131     }
132
133     if (ima_hash_algo_idx < 0) {
134         ...
135         if (ima_template_hash_algo == ima_hash_algo)
136             ima_template_hash_algo_idx = ima_hash_algo_idx;
137     }
138
139     if (ima_template_hash_algo_idx < 0)
140         ima_template_hash_algo_idx = NR_BANKS(ima_tpm_chip) + ima_extra_slots++;
141
142     ...
143
144     if (ima_template_hash_algo_idx >= NR_BANKS(ima_tpm_chip) && ↪
        ima_template_hash_algo_idx != ima_sha1_idx && ima_template_hash_algo_idx != ↪
        ima_hash_algo_idx) {
145         ima_algo_array[ima_template_hash_algo_idx].tfm = ↪
        ima_alloc_tfm(ima_template_hash_algo);
146         if (IS_ERR(ima_algo_array[ima_template_hash_algo_idx].tfm)) {
147             rc = PTR_ERR(ima_algo_array[ima_template_hash_algo_idx].tfm);
148             goto out_array;
149         }
150     }
151     ...
```

The new global variables, `ima_template_hash_algo` and `ima_template_hash_algo_idx`, have been declared as extern variables in the `ima.h` file, so that they can be accessed in other source files that include `ima.h`:

```
/* set during initialization */
extern int ima_template_hash_algo;
extern int ima_template_hash_algo_idx __ro_after_init;
...
```

Then, we introduced some changes in the functions that write the IMA measurement list in the MLs, `ima_measurements_show()` in binary format (defined in ima_fs.c, lines 127-187) and `ima_ascii_measurements_show()` in ASCII format (defined in ima_fs.c, lines 217-253). In both functions, `ima_sha1_idx` has been replaced with `ima_template_hash_algo_idx` in the statements where they refer to the template-hash to be written in the ML; moreover, the name of the hash algorithm has been added as prefix of the digest in the `ima_ascii_measurements_show()` function (listing C.9).

Listing C.9. Extract from ima_fs.c

```
/* print in ascii */
static int ima_measurements_show(struct seq_file *m, void *v)
{
    ...
    u32 template_hash_len;

    ...

    /* 2nd: template digest size */
    template_hash_len = !ima_canonical_fmt ? ↪
    hash_digest_size[ima_template_hash_algo] : ↪
    cpu_to_le32(hash_digest_size[ima_template_hash_algo]);
```

```
    ima_putc(m, &template_hash_len, sizeof(template_hash_len));

    /* 3rd: template digest */
    ima_putc(m, e->digests[ima_template_hash_algo_idx].digest, →
    hash_digest_size[ima_template_hash_algo]);

    ...
}

/* print in ascii */
static int ima_ascii_measurements_show(struct seq_file *m, void *v)
{
    ...

    /* 2nd: template hash */
    seq_printf(m, "%s:", hash_algo_name[ima_template_hash_algo]);
    ima_print_digest(m, e->digests[ima_template_hash_algo_idx].digest, →
    hash_digest_size[ima_template_hash_algo]);

    ...
}
```

Then, we added the possibility to select the default template-hash algorithm in the IMA Kconfig file, accessible in the IMA source code via the `CONFIG_IMA_DEFAULT_TEMPLATE_HASH` parameter (listing C.10).

Listing C.10. Extract from Kconfig

```
choice
        prompt "Default template-hash algorithm"
        default IMA_DEFAULT_TEMPLATE_HASH_SHA1
        depends on IMA
        help
                Select the default template-hash algorithm written in Measurement
                Log entries.  The compiled default template-hash algorithm can
                be overwritten using the kernel command line 'ima_template_hash='
                option.

        config IMA_DEFAULT_TEMPLATE_HASH_SHA1
                bool "SHA1 (default)"
                depends on CRYPTO_SHA1=y

        config IMA_DEFAULT_TEMPLATE_HASH_SHA256
                bool "SHA256"
                depends on CRYPTO_SHA256=y

        config IMA_DEFAULT_TEMPLATE_HASH_SHA512
                bool "SHA512"
                depends on CRYPTO_SHA512=y
endchoice

config IMA_DEFAULT_TEMPLATE_HASH
        string
        depends on IMA
        default "sha1" if IMA_DEFAULT_TEMPLATE_HASH_SHA1
        default "sha256" if IMA_DEFAULT_TEMPLATE_HASH_SHA256
        default "sha512" if IMA_DEFAULT_TEMPLATE_HASH_SHA512
```

Appendix D

Programmer's manual: Keylime REST APIs

This appendix describes the REST APIs exposed by the components of Keylime relative to version v6.0.0 and those added with the work of the thesis. All changes made to the framework are highlighted in the text by the character "*". Keylime APIs only accept JSON format and respond with a JSON object containing the following fields:

- `code` (int): the HTTP status code;
- `status` (string): textual explanation of the response status;
- `results` (JSON object): it contains specific data of the response and its format depends on the invoked API.

D.1 Cloud Agent

GET /v2/keys/pubkey (Unencrypted connection)

Retrieve Cloud Agent's NK_{pub} key, public part of an ephemeral rsa key used by Tenant and CV to encrypt U and V shares of the bootstrap key K_b.

Response JSON object

- `pubkey` (string): Cloud Agent's NK_{pub} key.

Example Response:

```
{
   "code": 200,
   "status": "Success",
   "results": {
                "pubkey": "-----BEGIN PUBLIC KEY---(...)---END PUBLIC KEY-----\n"
              }
}
```

POST /v2/keys/vkey (Unencrypted connection)

API used by the CV for sending the V share of the bootstrap key K_b, encrypted with NK_{pub} and base64 encoded, to the Cloud Agent.

Request JSON object

- `encrypted_key` (string): V share of K_b, encrypted with Cloud Agent's NK_{pub} and base64 encoded.

Example Request:

```
{
   "encrypted_key": "MN/F33jjuLiIuRH8fF7 ...  5foAqZCyZOAhQOONuWw==",
}
```

POST /v2/keys/ukey (Unencrypted connection)

API used by the Tenant to send the U share of the bootstrap key K_b, encrypted with NK_{pub} and base64 encoded, to the Cloud Agent, with optional payload encrypted with K_b.

Request JSON object

- `auth_tag` (string): it is the HMAC of Cloud Agent's UUID calculated with the bootstrap key K_b, used to by the Cloud Agent to check if K_b is correctly derived;
- `encrypted_key` (string): it is the U share of K_b, encrypted with NK_{pub} and base64 encoded;
- `payload` (string) - *optional*: it contains the user data, encrypted with K_b and base64 encoded.

Example Request:

```
{
  "auth_tag": "3876c08b30c ...  85de05c4c7cce",
  "encrypted_key": "iAckMZgZc8r43pF0iW8i ...  btLZBa9T+mmA==",
  "payload": "WcXpUr4G9yfvVaojNx6K2XZuDYRkFoZQhHrvZB+TKZqsq41g"
}
```

GET /v2/keys/verify (Unencrypted connection)

API used by the Tenant to verify that the Cloud Agent correctly derived the bootstrap key K_b.

Query parameters

- `challenge` (string): random string made up of 20 alphanumerical characters [a-Z,0-9].

Example Request:

```
/v2/keys/verify?challenge=78wAQ3cGHjx6103dKpfJ
```

Response JSON object

- `hmac` (string): it is the $HMAC_{K_b}(challenge)$; if the response is the correct one, the Tenant knows that the Cloud Agent correctly recomposed K_b.

Example Response:

```
{
   "code": 200,
   "status": "Success",
   "results": {
                "hmac": "719d992fb7 ...  dd7f9adee6c18"
              }
}
```

GET /v2/quotes/integrity (Unencrypted connection)

API used by the CV to get a new Integrity Report.

Query parameters

- `nonce` (string): random string made up of 20 alphanumerical characters [a-Z,0-9];
- `mask` (string): mask, expressed as hexadecimal number, used for specifying the PCRs to be included in the TPM quote;
- `vmask` (string): mask, expressed as hexadecimal number, used for specifying the vPCRs from the vTPM to be included in the deep-quote;
- `partial` (string): if set to "0", the NK_{pub} is added to the response; if set to "1", the key is not sent;
- `ml_seek`* (string) -*optional*: number representing the offset, in bytes, of the requested contents of the IMA ML; if not set, the IMA ML is sent entirely.

Example Request:

```
/v2/quotes/integrity?nonce=5yrFGxXz2qQljgg73bNd&mask=0x408000&partial=0&ml_seek=3250
```

Response JSON object

- `quote` (string): TPM quote;
- `hash_alg` (string): hash algorithm used by the TPM;
- `enc_alg` (string): encryption algorithm used by TPM;
- `sign_alg` (string): signing algorithm used by the TPM;
- `pubkey` (string) - *optional*: NK_{pub} key;
- `ima_measurement_list` (string) - *optional*: IMA ML contents starting from byte specified via `ml_seek`; it is included only if the `mask` query parameter specifies the IMA PCR;
- `mb_measurement_list` (string) - *optional*: it is the content, base64 encoded, of the measured boot event log; it is included only if the `mask` query parameter specifies PCR 0.

Example Response:

```
{
   "code": 200,
   "status": "Success",
   "results": {
                "quote": "reJz7H+Ls3i ...cGYRP+EwAUO3IUEAkA44Iwlw==",
                "hash_alg": "sha256",
                "enc_alg": "rsa",
                "sign_alg": "rsassa",
                "pubkey": "-----BEGIN PUBLIC KEY---(...)---END PUBLIC KEY-----\n",
                "ima_measurement_list": "10 sha256:3afc3490[...]  ima-dep-cgn ...  ",
                "mb_measurement_list": "fdrek73ggVtT48vXd..."
              }
}
```

GET /v2/quotes/identity (Unencrypted connection)

API used by the Tenant to get a TPM quote in order to verify the authenticity of the Cloud Agent's TPM. This quote is not used to verify the integrity state of the node, which is verified by the CV through the `GET /v2/quotes/integrity` API, so it does not contain PCRs.

Query parameters

- `nonce` (string): random string made up of 20 alphanumerical characters [a-Z,0-9].

Example Request:

```
/v2/quotes/identity?nonce=fYx07fjrE981QqjgF903
```

Response JSON object

- `quote` (string): TPM quote;
- `hash_alg` (string): hash algorithm used by the TPM;
- `enc_alg` (string): encryption algorithm used by TPM;
- `sign_alg` (string): signing algorithm used by the TPM;
- `pubkey` (string) - *optional*: NK_{pub} key.

Example Response:

```
{
   "code": 200,
   "status": "Success",
   "results": {
                "quote": "reJz7H+Ls3iDBoM...eN6OAWgAAgXUg0Q==",
                "hash_alg": "sha256",
                "enc_alg": "rsa",
                "sign_alg": "rsassa",
                "pubkey": "-----BEGIN PUBLIC KEY----- (...)-----END PUBLIC KEY-----\n"
              }
}
```

D.2 Cloud Verifier

GET /v2/agents (Mutual TLS connection)

Retrieve the list of all currently registered Cloud Agent UUIDs.

Response JSON object

- `uuids` (list[string]): it is the list of all UUIDs currently registered.

Example Response:

```
{
   "code": 200,
   "status": "Success",
   "results": {
                "uuids": [ "uuid1", "uuid2", ... ]
              }
}
```

GET /v2/agents/{agent_id:UUID} (Mutual TLS connection)

Retrieve the status of Cloud Agent *agent_id* from CV.

Response JSON object

- `operational_state` (int): current state of the Cloud Agent in the CV; it can be one of the following states:
  REGISTERED=0, START=1, SAVED=2, GET_QUOTE=3, GET_QUOTE_RETRY=4, PROVIDE_V=5, PROVIDE_V_RETRY=6, FAILED=7, TERMINATED=8, INVALID_QUOTE=9, TENANT_FAILED=10, UNKNOWN_CONTAINER*=11;
- `v` (string): V share of K_b for encrypted payload, base64 encoded (decoded length is 32 bytes);
- `ip` (string): Cloud Agent's IP address for the CV;
- `port` (int): Cloud Agent's port for the CV;
- `tpm_policy` (string): string-encoded JSON object containing the PCRs to be included in the TPM quote, with their corresponding whitelists, and the PCR mask to be sent to the Cloud Agent when a new Integrity Report is required;
- `vtpm_policy` (string): string-encoded JSON object specifying the vPCRs to be contained in the deep-quote, with their corresponding whitelists, and the vPCR mask to be sent to the Cloud Agent when a new Integrity Report is required;
- `meta_data` (string): metadata about the Cloud Agent; if the Cloud Agent has a certificate, it contains certificate's serial and subject; this object is sent by the Revocation Framework when this Cloud Agent fails its integrity state;
- `allowlist_len` (int): number of entries in the physical host's allowlist;
- `mb_refstate_len` (int): length of the measured boot reference state policy;
- `accept_tpm_hash_algs` (list[string]): accepted TPM hashing algorithms;
- `accept_tpm_encryption_algs` (list[string]): accepted TPM encryption algorithms;
- `accept_tpm_signing_algs` (list[string]): accepted TPM signing algorithms;
- `hash_alg` (string): used hashing algorithm;
- `enc_alg` (string): used encryption algorithm;
- `sign_alg` (string): used signing algorithm;
- `containers*` (JSON object): it specifies the containers associated to the Cloud Agent; each container is identified in the JSON object through its identifier and is itself a JSON object containing the following fields:
  - `operational_state` (int): current state of the container; it can be one of the following states: START=0 (it is waiting the first integrity check), TRUST=1, UNTRUST=2;
  - `allowlist` (JSON object): the container's allowlist which specifies, for each file-path, the list of reference digests;
  - `exclude` (list[string]): the container's exclude list, specifying a list of regular expressions for the file-paths that do not need to be attested;
  - `fnf` (list[string]) - *optional*: the list of file-paths not contained in the container's allowlist, detected during the remote attestation process;
  - `filehash_err` (list[string]) - *optional*: the list of file-paths whose hash did not match to those in the container's allowlist;

- `allow_unknown_containers`* (int): it is `1` if unknown containers are allowed on the host system, `0` otherwise;
- `unknown_containers`* (list[string]) - *optional*: the list of unknown containers' identifiers detected during the remote attestation process.

Example Response:


```
{
    "code": 200,
    "status": "Success",
    "results": {
                "operational_state": 11,
                "v": "yyNnlWwFRz1ZUzSe2YEpz9A5urtv6oywgttTF7VbBP4=",
                "ip": "127.0.0.1",
                "port": 9002,
                "tpm_policy": "{
                                \"0\":  [\"23BD73EC5A35...8BB8344CE77E3\"],
                                \"1\":  [\"73D3A3D17547...41134A2966268\"],
                                \"2\":  [\"B3BD342D6060...19A51CA37AB30\"],
                                ...
                                \"mask\":\"0x0007FF\"
                              }",
                "vtpm_policy": "{
                                 \"23\":  [\"ffffffffffff...ffffffffffff\", ...],
                                 \"15\":  [\"0000000000000...0000000000000\", ...],
                                 \"mask\":\"0x808000\"
                               }",
                "meta_data": "{
                               \"cert_serial\":7190667204669...9268666356441,
                               \"subject\":\"/C=US/ST=MA/...75BD81C00000\"
                             }",
                "allowlist_len": 0,
                "mb_refstate_len": 0,
                "accept_tpm_hash_algs": [ "sha512", "sha384", "sha256", "sha1" ],
                "accept_tpm_encryption_algs": [ "ecc", "rsa" ],
                "accept_tpm_signing_algs": [ "ecschnorr", "rsassa" ],
                "hash_alg": "sha256",
                "enc_alg": "rsa",
                "sign_alg": "rsassa",
                "containers": {
                                "125eef87c423": {
                                                  "operational_state": 1,
                                                  "allowlist": {
                                                     "f_path1": ["hash1", "hash2", ...],
                                                     "f_path2": ["hash3", "hash4", ...],
                                                      ...
                                                     },
                                                  "exclude": ["reg_ex1", "reg_ex2", ...],
                                                  "fnf": ["path1", "path2", ...],
                                                  "filehash_err": ["path3", "path4", ..]
                                                },
                                ...
                               }
                "allow_unknown_containers": 0,
                "unknown_containers": [ "5ffb2a71240c", "eafc6710be71" ]
            }
}
```


POST /v2/agents/{agent_id:UUID} (Mutual TLS connection)

Register a new Cloud Agent *agent_id* in the CV.

Request JSON object

- `v` (string): V share of K_b for encrypted payload, base64 encoded (decoded length is 32 bytes);
- `cloudagent_ip` (string): Cloud Agent's IP address for the CV;
- `cloudagent_port` (string): Cloud Agent's port for the CV;
- `tpm_policy` (string): (string): string-encoded JSON object containing the PCRs to be included in the TPM quote, with their corresponding whitelists, and the PCR mask to be sent to the Cloud Agent when a new Integrity Report is required;
- `vtpm_policy` (string): string-encoded JSON object specifying the vPCRs to be contained in the deep-quote, with their corresponding whitelists, and the vPCR mask to be sent to the Cloud Agent when a new Integrity Report is required;
- `metadata` (string): metadata about the Cloud Agent; if the Cloud Agent has a certificate, it contains the certificate's serial and subject; this object is sent by the Revocation Framework when the Cloud Agent is evaluated untrusted;
- `allowlist` (string): string-encoded JSON object containing the physical host's allowlist and exclude list;
- `mb_refstate` (string): measured boot reference state policy;
- `ima_sign_verification_keys` (string): string-encoded list of IMA signature verification public keys;
- `revocation_key` (string): rsa private Key which the CV has to use for signing the revocation message of this Cloud Agent;
- `accept_tpm_hash_algs` (list[string]): accepted TPM hashing algorithms;
- `accept_tpm_encryption_algs` (list[string]): accepted TPM encryption algorithms;
- `accept_tpm_signing_algs` (list[string]): accepted TPM signing algorithms;
- `containers`* (JSON object) - *optional*: it specifies the containers associated to the Cloud Agent; each container is identified in the JSON object through its identifier and is associated to JSON object containing the following fields:
  - `allowlist` (JSON object): the container's allowlist which specifies, for each file-path, the list of reference digests;
  - `exclude` (list[string]): the container's exclude list, which specifies a list of regular expressions for the file-paths that do not need to be attested;
- `allow_unknown_containers`* (int) - *optional*: if it is set to `1`, unknown containers will be allowed on the agent's host system; if it is set to `0`, or the field is absent, unknown containers will cause an integrity failure during attestation.

Example Request:

```
{
   "v": "3HZMmIEc6yyjfoxdCwcOgPk/6X1GuNG+tlCmNgqBM/I=",
   "cloudagent_ip": "127.0.0.1",
   "cloudagent_port": 9002,
   "tpm_policy": "{
                     \"0\":  [\"23BD73EC5A35C...8BB8344CE77E3\"],
                     \"1\":  [\"73D3A3D17547D...41134A2966268\"],
                     \"2\":  [\"B3BD342D6060F...19A51CA37AB30\"],
                      ...
                     \"mask\":\"0x0007FF\"
                 }",
```

```
    "vtpm_policy": "{
                      \"23\":  [\"ffffffffffff...ffffffffffff\", ...  ],
                      \"15\":  [\"0000000000000...0000000000000\", ...],
                      \"mask\":  \"0x808000\"
                    }",
    "metadata": "{
                   \"cert_serial\":7190667204669...0742724395900,
                   \"subject\":  \"/C=US/ST=MA/...75BD81C00000\"
                 }",
    "allowlist": "{
                    \"allowlist\":{
                                    \"boot_aggregate\":[\"hash1\", \"hash2\", ...],
                                    \"file_path1\":[\"hash3\", \"hash4\", ...],
                                    \"file_path2\":[\"hash5\", \"hash6\", ...],
                                    ...
                                  },
                   \"exclude\":[\"reg_ex1\", \"reg_ex2\", ...]
                  }",
    "mb_refstate": "null",
    "ima_sign_verification_keys": "[]",
    "revocation_key": "-----BEGIN PRIVATE KEY----- (...)-----END PRIVATE KEY-----\n",
    "accept_tpm_hash_algs": [ "sha512", "sha384", "sha256", "sha1" ],
    "accept_tpm_encryption_algs": [ "ecc", "rsa" ],
    "accept_tpm_signing_algs": [ "ecschnorr", "rsassa" ],
    "containers": {
                    "dc912aec76bd": {
                                       "allowlist": {
                                                       "f_path1": ["hash1", "hash2", ...],
                                                       "f_path2": ["hash3", "hash4", ...],
                                                        ...
                                                     },
                                       "exclude": ["reg_ex1", "reg_ex2", ...]
                                    },
                    "ffbec71a025c": {
                                       "allowlist": {
                                                       "f_path1": ["hash1", "hash2", ...],
                                                       "f_path2": ["hash3", "hash4", ...],
                                                         ...
                                                     },
                                       "exclude": ["reg_ex1", "reg_ex2", ...]
                                    },
                       ...
                    },
    "allow_unknown_containers": 0
}
```

Response JSON object

- `not_accepted_contIDs`* (list[string]) - *optional*: it is the list of the identifiers of those containers that have not been registered in the CV; this happens one the exclude list of a container is misformatted.

Example Responses:

```
{
   "code": 200,
   "status": "Success",
   "results": {}
}

{
   "code": 202,
   "status": "Partially accepted",
   "results": {
```

```
                "not_accepted_contIDs": [ "e47cbfe9120c", "9cd41cef871c", ... ]
            }
}
```

DELETE /v2/agents/{agent_id:UUID} (Mutual TLS connection)

Remove the Cloud Agent *agent_id* from the CV database.

Response JSON object
Example Responses:

```
{
   "code": 200,
   "status": "Success",
   "results": {}
}
```

Response 200 is returned when the Cloud Agent {agent_id} has been removed from the CV's database; the deletion is successfully completed only if the agent's `operational_state` is one of the following: SAVED, FAILED, TERMINATED, TENANT_FAILED, INVALID_QUOTE.

```
{
   "code": 202,
   "status": "Accepted",
   "results": {}
}
```

Response 202 is returned when the deletion request is sent for an agent whose `operational_state` is GET_QUOTE. Upon receiving the deletion request, the `operational_state` of the Cloud Agent {agent_id} is set to TERMINATED and it will be removed from the CV's database when it will be scheduled for the next remote attestation.

PUT /v2/agents/{agent_id:UUID}/reactivate (Mutual TLS connection)

Restart the RA process for the Cloud Agent {agent_id}.

Response JSON object
Example Responses:

```
{
   "code": 200,
   "status": "Success",
   "results": {}
}
```

Response 200 means that the `operational_state` of the Cloud Agent {agent_id} is set to START and the agent is re-inserted in a "loop" for periodic attestation.

```
* {
     "code": 403,
     "status": "Forbidden",
     "results": {}
  }
```

Response with code 403 has been added to the API and means that the Cloud Agent {agent_id} is already in an active state; this check has been added to prevent a node already inserted in the queue of nodes to be attested from being inserted several times.

PUT /v2/agents/{agent_id:UUID}/stop (Mutual TLS connection)

Stop the RA process for the Cloud Agent {agent_id}, but do not delete the agent from the CV database. The `operational_state` of the Cloud Agent {agent_id} is set to TENANT_FAILED; this will cause that the CV will not query new IRs to this agent until it is reactivated.

PUT /v2/agents/{agent_id:UUID}/allow_unknown_containers * (Mutual TLS connection)

Allow / disallow the execution of unknown containers on host system of the Cloud Agent {agent_id}.

Request JSON object

- `allow_unknown_containers` (int): if set to 0, unknown containers running on the host system will cause an integrity failure; if set to 1, unknown containers will be admitted.

Example Request:

```
{
   "allow_unknown_containers": 1
}
```

GET /v2/agents/{agent_id:UUID}/allowlist * (Mutual TLS connection)

Get the current host system's allowlist associated to the Cloud Agent {agent_id}.

Response JSON object

- `allowlist` (JSON object): host system allowlist.

Example Response:

```
{
   "code": 200,
   "status": "Success",
   "results": {
                "allowlist": {
                                "file_path1": ["hash1", "hash2", ...],
                                "file_path2": ["hash3", "hash4", ...],
                                  ...
                              }
              }
}
```

PUT /v2/agents/{agent_id:UUID}/allowlist * (Mutual TLS connection)

Substitute the host system's allowlist associated to the Cloud Agent {agent_id} with the one provided in the request body.

Request JSON object

- `allowlist` (JSON object): new host system's allowlist.

Example Request:

```
{
   "allowlist": {
                  "boot_aggregate": ["hash1", "hash2", ...],
                  "file_path1": ["hash3", "hash4", ...],
                  "file_path2": ["hash5", "hash6", ...],
                   ...
                }
}
```

PATCH /v2/agents/{agent_id:UUID}/allowlist * (Mutual TLS connection)

Patch the host system's allowlist, associated to the Cloud Agent {agent_id}.

Request JSON object

- `delete_paths` (list[string]) - *optional field*: list of file-paths that will be removed from the host system's allowlist;
- `put_paths` (JSON object) - *optional field*: file-paths that will be inserted or updated in the host system's allowlist.

Example Request:

```
{
   "delete_paths": [ "file_path1", "file_path2", ... ],
   "put_paths": {
                  "file_path3": [ "hash1", "hash2", ... ],
                  "file_path4": [ "hash3", "hash4", ... ],
                    ...
                }
}
```

Response JSON object

- `allowlist`: the entire host system's allowlist newly updated.

```
{
   "code": 200,
   "status": "Success",
   "results":  {
                "allowlist": {
                                "boot_aggregate": [ "hash1", "hash2", ... ],
                                "file_path1": [ "hash3", "hash4", ... ],
                                "file_path2": [ "hash5", "hash6", ... ],
                                  ...
                              }
                }
}
```

GET /v2/agents/{agent_id:UUID}/exclude * (Mutual TLS connection)

Get the current host system's exclude list associated to the Cloud Agent {agent_id}.

Response JSON object

- `exclude` (list[string]): host system's exclude list.

Example Response:

```
{
   "code": 200,
   "status": "Success",
   "results": {
               "exclude": [ "reg_ex1", "reg_ex2", ... ]
             }
}
```

PUT /v2/agents/{agent_id:UUID}/exclude * (Mutual TLS connection)

Substitute the host system's exclude list associated to the Cloud Agent {agent_id} with the one provided in the request body. To reset the exclude list, provide an empty list in the request body.

Request JSON object

- `exclude` (list[string]): the new host system's exclude list.

Example Request:

```
{
   "exclude": [ "reg_ex1", "reg_ex2", ... ]
}
```

Response JSON object

- `exclude` (list[string]): the newly updated host system's exclude list.

Example Responses:

```
{
   "code": 200,
   "status": "Success",
   "results":  {
                 "exclude": [ "reg_ex1", "reg_ex2", ... ]
               }
}

{
   "code": 400,
   "status": " Exclude list regex is misformatted.  Please correct the issue and try
    again.",
   "results":  {}
}
```

Response code `400` can occur if some of the regular exception provided in the request body is not valid.

GET /v2/agents/{agent_id:UUID}/containers * (Mutual TLS connection)

Retrieve the list of all container identifiers currently registerd with the Cloud Agent {agent_id}.

Response JSON object

- `uuid` (string): Cloud Agent's UUID;
- `cont_ids` (list[string]): list of container identifiers registered in the Cloud Agent.

Example Response:

```
{
   "code": 200,
   "status": "Success",
   "results": {
                "uuid":  "UUID1",
                "cont_ids": [ "a02ddec7129c", "23190cbdeefa", ... ]
              }
}
```

POST /v2/agents/{agent_id:UUID}/containers * (Mutual TLS connection)

Add the containers specified in the request body to the list of containers currently registerd with the Cloud Agent {agent_id}.

Request JSON object

The request body is a dictionary of container identifiers, each of which is associated to a JSON object with the following fields:

- `allowlist` (JSON object) - *optional*: the container's allowlist;
- `exclude` (list[string]) - *optional*: the container's exclude list.

Example Request:

```
{
   "67decf291bbe": {
                     "allowlist": {
                                    "file_path1": [ "hash1", "hash2", ... ],
                                    "file_path2": [ "hash3", "hash4", ... ],
                                     ...
                                  },
                     "exclude": ["reg_ex1", "reg_ex2", ...]
                   },
   "c29da875e2fb": {
                     "allowlist": {
                                    "file_path1": [ "hash1", "hash2", ... ],
                                    "file_path2": [ "hash3", "hash4", ... ],
                                     ...
                                  },
                     "exclude": [ "reg_ex1", "reg_ex2", ... ]
                   },
   ...
}
```

Response JSON object

Example Responses:

```
{
   "code": 200,
   "status": "Success",
   "results": {
                "1172cd64feed": {
                                  "operational_state": 2,
                                  "allowlist": {
                                                 "f_path1": [ "hash1", "hash2", ... ],
                                                 "f_path2": [ "hash3", "hash4", ... ],
                                                 ...
```

```
                                    },
                                "exclude": [ "reg_ex1", "reg_ex2", ... ],
                                "fnf": [ "fnf_path1", "fnf_path2", ... ],
                                "filehash_err": [ "path_1", "path_2", ... ]
                            },
                "feca384f5412": { ... },
                "67decf291bbe": { ... },
                "c29da875e2fb": { ... }
            }
}
```

When response code is 200, the response body contains the list of all containers registered in the Cloud Agent {agent_id}, newly updated with the containers provided in the request body.

```
{
   "code": 400,
   "status": "Expected non zero content length",
   "results": {}
}

{
   "code": 400,
   "status": "Container ID " + contID + " - Invalid regex:  " + regex_err.msg + ".",
   "results": {}
}
```

Response code 400 occurs when the request body is empyt or one of the containers specified in the request body has an invalid regular expression in its "exclude" list field.

```
{
   "code": 409,
   "status": "The container " + contID + " is already registered in agent " +
    {agent_id},
   "results": {}
}
```

Response code 409 occurs when one of the containers listed in the request body is already registered in the Cloud Agent {agent_id}.

PUT /v2/agents/{agent_id:UUID}/containers * (Mutual TLS connection)

Substitute the list of containers registered in the Cloud Agent {agent_id} with the one provided in the request body. If some container has not valid information associated in the req. body, then it will be ignored. In order to remove all containers registered in the Cloud Agent, send a request body with no container identifier.

Request JSON object

The request body is a dictionary of container identifiers, each of which is associated to a JSON object with the following fields:

- `allowlist` (JSON object) - *optional*: the container's allowlist;
- `exclude` (list[string]) - *optional*: the container's exclude list.

Example Request:

```
{
   "fc2704decbf3": {
                    "allowlist": {
                                    "file_path1": [ "hash1", "hash2", ...],
```

```
                                    "file_path2": [ "hash3", "hash4", ... ],
                                       ...
                                  },
                        "exclude": [ "reg_ex1", "reg_ex2", ... ]
                    },
    "ed62bf75aecb": { ... }
    ...
}
```

Response JSON object

Example Responses:

```
{
    "code": 200,
    "status": "Success",
    "results": {
                "fc2704decbf3": {
                                  "operational_state": 0,
                                  "allowlist": {
                                                "f_path1": ["hash1", "hash2", ...],
                                                "f_path2": ["hash3", "hash4", ...],
                                               ...
                                              },
                                  "exclude": ["reg_ex1", "reg_ex2", ...]
                                },
                "ed62bf75aecb": { ... },
                    ...
              }
}
```

GET /v2/agents/{agent_id:UUID}/containers/{container_id} *

(Mutual TLS connection)

Retrieve information about the container {container_id} associated to the Cloud Agent {agent_id}.

Response JSON object

- `container` (JSON object): it contains the information related to the {container_id}.

Example Response:

```
{
    "code": 200,
    "status": "Success",
    "results": {
                "container": {
                                "contID": "b9aef4312bcc",
                                "operational_state": 2,
                                "allowlist": {
                                              "file_path1": [ "hash1", "hash2", ... ],
                                              "file_path2": [ "hash3", "hash4", ... ],
                                                ...
                                             },
                                "exclude": [ "reg_ex1", "reg_ex2", ... ],
                                "fnf": [ "path_fnf1", "path_fnf2", ... ],
                                "filehash_err": [ "path_1", "path_2", ... ]
                              }
              }
}
```

PUT /v2/agents/{agent_id:UUID}/containers/{container_id} *
(Mutual TLS connection)

Update the allowlist and exclude list of the {container_id} associated to the Cloud Agent {agent_id}; if the {container_id} is not registered, add it to the list of containers associated to {agent_id}.

Request JSON object

- `allowlist` (JSON object): the allowlist of the {container_id};
- `exclude` (list[string]): the exclude list of the {container_id};

Example Request:

```
{
   "allowlist": {
                  "file_path1": [ "hash1", "hash2", ... ],
                  "file_path2": [ "hash3", "hash4", ... ],
                    ...
                },
   "exclude": [ "reg_ex1", "reg_ex2", .... ]
}
```

Response JSON object

Example Response:

```
{
   "code": 201,
   "status": "Created",
   "results": {
                "contID": "c91e447a05cb",
                "operational_state": 0,
                "allowlist": {
                               "file_path1": [ "hash1", "hash2", ... ],
                               "file_path2": [ "hash3", "hash4", ... ],
                                 ...
                             },
                "exclude": [ "reg_ex1", "reg_ex2", ... ]
              }
}
```

If the {container_id} has been added, the response code is `201` "Created", if it has been updated, the response code is `200` "Success".

DELETE /v2/agents/{agent_id:UUID}/containers/{container_id} *
(Mutual TLS connection)

Remove the {container_id} from the Cloud Agent {agent_id}.

GET /v2/agents/{agent_id:UUID}/containers/{container_id}/allowlist *
(Mutual TLS connection)

Get the current allowlist of {container_id}, registered in the CV and associated to the Cloud Agent {agent_id}.

Response JSON object

- `allowlist` (JSON object): the current {container_id}'s allowlist.

Example Response:

```
{
   "code": 200,
   "status": "Success",
   "results": {
                "allowlist": {
                                "file_path1": [ "hash1", "hash2", ... ],
                                "file_path2": [ "hash3", "hash4", ... ],
                                  ...
                              }
              }
}
```

PUT /v2/agents/{agent_id:UUID}/containers/{container_id}/allowlist *

(Mutual TLS connection)

Substitute the allowlist of {container_id}, associated to the Cloud Agent {agent_id}, with the one provided in the request body.

Request JSON object

- `allowlist` (JSON object): the new allowlist for {container_id}.

Example Request:

```
{
   "allowlist": {
                  "/path/to/file1": [ "hash1", "hash2", ... ],
                  "/path/to/file2": [ "hash3", "hash4", ... ],
                    ...
                }
}
```

PATCH /v2/agents/{agent_id:UUID}/containers/{container_id}/allowlist *

(Mutual TLS connection)

Patch the allowlist of {container_id}, registered in the CV and associated to the Cloud Agent {agent_id}.

Request JSON object

- `delete_paths` (list[string]) - *optional* : the list of file paths to be removed from the {container_id} allowlist;
- `put_paths` (JSON object) - *optional* : the file paths to be added or updated in the {container_id} allowlist.

Example Request:

```
{
   "delete_paths": [ "/path/to/file1", "/path/to/file2", ... ],
   "put_paths": {
                   "/path/to/file3": [ "hash3", "hash4", ... ],
                   "/path/to/file4": [ "hash5", "hash6", ... ],
                      ...
                }
}
```

Response JSON object

- `allowlist` (JSON object): the newly updated {container_id} allowlist.

Example Responses:

```
{
   "code": 200,
   "status": "Success",
   "results":  {
                 "allowlist": {
                                 "/path/to/file3": [ "hash3", "hash4", ... ],
                                 "/path/to/file4": [ "hash5", "hash6", ... ],
                                    ...
                               }
               }
}

{
   "code": 400,
   "status": "The request body does not have the fields \"delete_paths\" or
    \"put_paths\"",
   "results": {}
}
```

Response code 400 can occur if request body is empty or does not contain neither `delete_paths` nor `put_paths`.

GET /v2/agents/{agent_id:UUID}/containers/{container_id}/exclude *

(Mutual TLS connection)

Get the current exclude list of {container_id}, registered in the CV and associated to the Cloud Agent {agent_id}.

Response JSON object

- `exclude` (list[string]): the {container_id}'s exclude list.

Example Response:

```
{
   "code": 200,
   "status": "Success",
   "results": {
                 "exclude": [ "reg_ex1", "reg_ex2", ... ]
              }
}
```

PUT /v2/agents/{agent_id:UUID}/containers/{container_id}/exclude *

(Mutual TLS connection)

Substitute the exclude list of {container_id}, associated to the Cloud Agent {agent_id}, with the one provided in the request body.

Request JSON object

- `exclude` (list[string]): the new exclude list for the {container_id}.

Example Request:

```
{
   "exclude": [ "reg_ex1", "reg_ex2", ... ]
}
```

Response JSON object

- `exclude` (list[string]): the newly updated exclude list of {container_id}.

Example Responses:

```
{
   "code": 200,
   "status": "Success",
   "results": {
                "exclude": [ "reg_ex1", "reg_ex2", ... ]
              }
}

{
   "code": 400,
   "status": "Invalid regex:  " + regex_err.msg + ".",
   "results": {}
}
```

Response code `400` occurs when one of the regular expressions in the request body is not valid, the request body is empty or does not contain the `exclude` field.

D.3 Registrar

GET /v2/agents (Mutual TLS connection)

Retrieve the list of all Cloud Agents currently registered.

Response JSON object

- `uuids` (list[string]): list of all Cloud Agent UUIDs currently registered.

Example Response:

```
{
   "code": 200,
   "status": "Success",
   "results": {
                "uuids":  [  "uuid1", "uuid2", ... ]
              }
}
```

GET /v2/agents/{agent_id:UUID} (Mutual TLS connection)

API used by Tenant and CV to retrieve TPM credentials of Cloud Agent {agent_id}, needed to validate TPM quotes.

Response JSON object

- `aik_tpm` (string): base64 encoded AIK_{pub}, with format TPM2B_PUBLIC from tpm2-tss;

- `ek_tpm` (string): base64 encoded EK_{pub}; when the Cloud Agent runs directly on the physical host, it will submit `ekcert` and `ek_tpm` will be the public key of that certificate;
- `ekcert` (string): base64 encoded EK certificate in DER format;
- `regcount` (int): counter of the number of registrations made by the same Cloud Agent;
- `provider_keys` (JSON object) -*optional*: this field is present only if the Cloud Agent runs in a VM, in which case previous fields refer to Cloud Agent's vTPM (`ekcert` is `None` or "emulator" in this case), while fields inside `provider_keys` refer to the provider's physical TPM:
  - `aik` (string): physical TPM's AIK_{pub};
  - `ek` (string): physical TPM's EK_{pub};
  - `ekcert` (string): physical TPM's EK certificate;
  - `regcount` (int): registration counter.

Example Responses:

```
{
   "code": 200,
   "status": "Success",
   "results":  {
                "aik_tpm": "ARgAAQALAAUAcgAAABAAF...4FlbNOAW3APH8c+jZ3tgbt",
                "ek_tpm": "AToAAQALAAMAsgAgg3GXZ...mJx63obCqx9z5BltV5YQ==",
                "ekcert": "MIIEGTCCAoGgAwIBAgIBBTA...y2z8m7UHiLCbamSe6m7W",
                "regcount": 1,
                "provider_keys": {
                                   "aik": "fdshiuOHF+...fjdsk670",
                                   "ek": "FD94HOhjkd...4Jjkd8jd",
                                   "ekcert": "dfslLJ897...JK83ijdJ",
                                   "regcount": 1
                                  }
              }
}

{
   "code": 404,
   "status": "agent_id not yet active",
   "results": {}
}
```

Response code `404` can happen if the Cloud Agent {agent_id} does not exist in the Registrar or if it has not yet completed the registration procedure.

DELETE /v2/agents/{agent_id:UUID} (Mutual TLS connection)

Remove the Cloud Agent {agent_id} from Registrar's database.

POST /v2/agents/{agent_id:UUID} (Unencrypted connection)

API used by the Cloud Agent to post physical TPM's credentials if it runs directly on the physical host, vTPM's credential if it runs in a VM.

Request JSON object

- `ek_tpm` (string) - *optional*: base64 encoded EK_{pub}; this field is present only if `ekcert` is `None` or "emulator";

- `ekcert` (string): base64 encoded EK certificate in DER format;
- `aik_tpm` (string): base64 encoded AIK_{pub}, with format TPM2B_PUBLIC from tpm2-tss;

Example Request:

```
{
  "ekcert": "MIIEGTCCAoGgAw...2z8m7UHiLCbamSe6m7W",
  "aik_tpm": "ARgAAQALAAUAc...9lZmMvregrFHKYc7CXChz"
}
```

Response JSON object

- `blob` (string): base64 encoded blob containing the `aik_tpm` name and a challenge K_e, encrypted with EK_{pub}, that is `ek_tpm` or the public key contained in `ekcert`.

Example Response:

```
{
   "code": 200,
   "status": "Success",
   "results": {
                "blob": "utzA3gAAAAEARAAgC/w9...NTByalxbulg8x1eGtZyuQF"
              }
}
```

PUT /v2/agents/{agent_id:UUID}/activate (Unencrypted connection)

API used by the Cloud Agent, when it runs on physical host, to respond to the challenge sent by the Registrar and activate the registration.

Request JSON object

- `auth_tag` (string): HMAC of the {agent_id} calculated with the challenge K_e: $HMAC_{K_e}(agent\_id)$.

Example Request:

```
{
   "auth_tag": "7087ba88746886262de7...55ddb96e32efdd8745d0bdfef"
}
```

PUT /v2/agents/{agent_id:UUID}/vactivate (Unencrypted connection)

API used by the Cloud Agent, when it runs in a VM, to respond to the challenge sent by the Registrar and activate the registration.

Request JSON object

- `deepquote` (string): challenge response, consisting of a deep-quote signed with physical AIK_{priv} and $vAIK_{priv}$ and created with "nonce" equals to the hash of the challenge K_e; it contains vPCR 16 with the extension of {agent_id}, $vAIK_{pub}$ and vEK_{pub}:

$$DeepQuote_{AIK\_vAIK}(H(K_e), v16 : H(agent\_id, vAIK_{pub}, vEK_{pub}))$$

Example Request:

```
{
   "deepquote": "reJz7H+Ls3iD...O3IUEAkA44Iwlw=="
}
```

D.4 Tenant Webapp

GET /v2/webapp/ (Server-authenticated TLS connection)

Get HTML web page.

GET /v2/logs/tenant (Server-authenticated TLS connection)

Get the Tenant log.

Query Parameter

- `pos` (int) - *optional*: it specifies the start-line in the log file to be sent.

Example Request:

```
/v2/logs/tenant?pos=20
```

Response JSON object

- `log` (list[string]): content of Tenant's log file.

```
{
   "code": 200,
   "status": "Success",
   "results": {
                "log": [ "line_pos", "line_pos+1", "line_pos+2", ... ]
              }
}
```

GET /v2/agents/ (Server-authenticated TLS connection)

Get the ordered list of registerd Cloud Agents.

Response JSON object

- `uuids` (list[string]): UUIDs of Cloud Agents registered in Keylime, ordered on the basis of the agent's `operational_state`; the choosed `operational_state` ordering is the following one:
  1) TENANT_FAILED, 2) INVALID_QUOTE, 3) FAILED, 4) UNKNOWN_CONTAINER, 5) GET_QUOTE, 6) GET_QUOTE_RETRY, 7) PROVIDE_V, 8) PROVIDE_V_RETRY, 9) SAVED, 10) START, 11) TERMINATED, 12) REGISTERED.

```
{
   "code": 200,
   "status": "Success",
   "results": {
                "uuids": [ "uuid1", "uuid2", ... ]
              }
}
```

GET /v2/agents/{agent_id:UUID} (Server-authenticated TLS connection)

Get details about Cloud Agent {agent_id}.

Response JSON object

The response body has the same format as that returned by the API `GET /v2/agents/{agent_id:UUID}` of the CV, already described above, with the addition of the field `id` containing the {agent_id}.

PUT /v2/agents/{agent_id:UUID} (Server-authenticated TLS connection)

Reactivate Cloud Agent {agent_id}; it internally invokes the API, exposed by the CV, `PUT /v2/agents/{agent_id:UUID}/reactivate`.

POST /v2/agents/{agent_id:UUID} (Server-authenticated TLS connection)

Register a new Cloud Agent {agent_id} in CV e provide an encrypted payload to the {agent_id}.

Request JSON object

- `agent_ip` (string): Cloud Agent's contact IP address;
- `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, to be 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 with `keyfile_data`;
- `keyfile_data` (string) - *optional*: "base64_Kb\nbase64_U\nbase64_V"; this field is used when `ptype` = KEYFILE and contains the bootstrap key K_b used to cipher `file_data`, together with the U and V keys in which K_b has been splitted;
- `include_dir_data` (list[string]) - *optional*: ["data1_base64", "data2_base64", ...]; this field is used when `ptype` = CA_DIR and contains a list of data to be included in a zip file that will be sent to the Cloud Agent; the zip file will include also the certificates contained in the directory `ca_dir`, among which there is the RevocationNotifier-cert.crt used by the Cloud Agent to check the authenticity of revocation messages, and will be sent to the agent encrypted with a random bootstrap key K_b choosen by the Tenant;
- `include_dir_name` (list[string]) - *optional*: ["name1", "name2", ...]; this field is used when `ptype` = CA_DIR, it is a list of names corresponding to the `include_dir_data` list; the length of this list has to be equal to that of the `include_dir_data` list;
- `ca_dir` (string) - *optional*: this field is used when `ptype` = CA_DIR and is the path of the CA directory on the machine where the Tenant Webapp is running; if set to "default", the path used is `/var/lib/keylime/ca`;
- `ca_dir_pw` (string) - *optional*: this field is used when `ptype` = CA_DIR and contains the CA password;
- `tpm_policy` (JSON object) - *optional*: it specifies the PCRs to be requested in the TPM quote and their whitelist, with the exception of IMA PCR, which is added to the PCR mask if `a_list_data`, `a_list_url` or `ima_sign_verification_keys` are specified;

- `vtpm_policy` (JSON object) - *optional*: it specifies the vPCRs to be requested in the deep-quote and their whitelists, with the exception of IMA vPCR, which is added to the vPCR mask if `a_list_data`, `a_list_url` or `ima_sign_verification_keys` are specified;
- `a_list_data` (list[string]) - *optional*: it is the whitelist for the host system;
- `a_list_url`* (string) - *optional*: it is the URL to download the host's whitelist from; if this field is specified, `a_list_data` can not be specified;
- `e_list_data` (list[string]) - *optional*: it is a list of regular expressions used to match files that has to be excluded during host system attestation; empty strings or strings starting with "#" will be removed from the list;
- `ima_sign_verification_keys` (list[string]) - *optional*: it is the IMA public keyring;
- `mb_refstate` (list[string]) - *optional*: Measured Boot reference state policy;
- `containers`* (JSON object) - *optional*: it specifies the containers associated to the Cloud Agent; each container is identified in the JSON object through its identifier and is associated to JSON object containing the following fields:
  - `a_list_data` (list[string]): it is the container's whitelist, specified as a list of strings in the format "hash file-path";
  - `a_list_url` (string): it is the URL to download the container's whitelist from; if this field is specified, `a_list_data` can not be specified for that container;
  - `e_list_data` (list[string]): it is the container's exclude list, which specifies a list of regular expressions for the file-paths that do not need to be attested;
- `allow_unknown_containers`* (int) - *optional*: if it is set to `1`, unknown containers will be allowed on the Cloud Agent's host system; if it is set to `0`, or the field is absent, unknown containers will cause an integrity failure during attestation.

Example Request:

```
{
   "agent_ip" : "127.0.0.1",
   "ptype": 0,
   "file_data" : "base64",
   "tpm_policy": {
                  "0": ["23BD73EC5A35CB4...20D8BB8344CE77E3"],
                  "1": ["73D3A3D17547D7B...8E841134A2966268"],
                  "2": ["B3BD342D6060FFE...FEC19A51CA37AB30"],
                   ...
                 },
   "a_list_data": [
                   "hash0_sha256  boot_aggregate",
                   "hash1_sha1    /path/to/file1",
                   "hash2_sha256  /path/to/file1",
                   "hash3_sha1    /path/to/file2",
                   "hash4_sha256  /path/to/file2",
                      ...
                  ],
   "e_list_data": [ "/var/log/wtmp", "/root/etc/fstab", "/sys/fs/.*", ... ],
   "allow_unknown_containers": 0,
   "containers": {
                  "5cb9ae56bc24": {
                                    "a_list_data": [
                                                     "hash1_sha1    /path/to/file1",
                                                     "hash2_sha256  /path/to/file1",
                                                     "hash3_sha1    /path/to/file2",
                                                     "hash4_sha256  /path/to/file2",
                                                        ...
                                                   ],
```

```
                                    "e_list_data": [ "reg_ex1", "reg_ex2", ... ]
                                  },
                    "dc5710bc5eed": { ... }
                  }
}
```

DELETE /v2/agents/{agent_id:UUID} (Server-authenticated TLS connection)

Remove the Cloud Agent {agent_id} from the CV; the agent still remains registered in the Registrar.

PUT /v2/agents/{agent_id:UUID}/allow_unknown_containers * (Server-authenticated TLS connection)

Allow / disallow the execution of unknown containers on host system of the Cloud Agent {agent_id}. This API will internally invoke the corresponding API of the CV, presented above.

Request JSON object

- `allow_unknown_containers` (int): if set to 0, unknown containers running on the host system will cause an integrity failure; if set to 1, unknown containers will be admitted.

Example Request:

```
{
   "allow_unknown_containers": 1
}
```

GET /v2/agents/{agent_id:UUID}/allowlist * (Server-authenticated TLS connection)

Get the current host system's allowlist associated to the Cloud Agent {agent_id}. It internally invokes the corresponding API exposed by the CV.

Response JSON object

- `allowlist` (JSON object): host system allowlist.

Example Response:

```
{
   "code": 200,
   "status": "Success",
   "results": {
                  "allowlist": {
                                    "/path/to/file1": ["hash1", "hash2", ...],
                                    "/path/to/file2": ["hash3", "hash4", ...],
                                     ...
                                }
              }
}
```

PUT /v2/agents/{agent_id:UUID}/allowlist * (Server-authenticated TLS connection)

Substitute the host system's allowlist associated to the Cloud Agent {agent_id} with the one provided in the request body. This API will internally invoke the corresponding one exposed by the CV.

Request JSON object

- `allowlist` (JSON object): new host system's allowlist.

Example Request:

```
{
   "a_list_data": [
                    "hash1_sha1    boot_aggregate",
                    "hash2_sha256  boot_aggregate",
                    "hash3_sha1    /path/to/file1",
                    "hash4_sha256  /path/to/file1",
                    "hash5_sha1    /path/to/file2",
                    "hash6_sha256  /path/to/file2",
                      ...
                  ]
}
```

PATCH /v2/agents/{agent_id:UUID}/allowlist * (Server-authenticated TLS connection)

Patch the host system's allowlist, associated to the Cloud Agent {agent_id}; this API internally invokes the corresponding one expose by the CV.

Request JSON object

- `delete_paths` (list[string]) - *optional field*: list of file-paths that will be removed from the host system's allowlist;
- `put_paths` (list[string]) - *optional field*: file-paths that will be inserted or updated in the host system's allowlist.

Example Request:

```
{
   "delete_paths": [ "/path/to/file1", "/path/to/file2", ... ],
   "put_paths": [
                  "hash3_sha1    /path/to/file3",
                  "hash4_sha256  /path/to/file3",
                  "hash5_sha1    /path/to/file4",
                  "hash6_sha256  /path/to/file4",
                    ...
                ]
}
```

Response JSON object

- `allowlist`: the newly updated host system's allowlist.

```
{
   "code": 200,
   "status": "Success",
   "results":  {
```

```
                "allowlist": {
                                "boot_aggregate": [ "hash1_sha1", "hash2_sha256" ],
                                "/path/to/file3": [ "hash3_sha1", "hash4_sha256" ],
                                "/path/to/file4": [ "hash5_sha1", "hash6_sha256" ],
                                  ...
                              }
            }
}
```

GET /v2/agents/{agent_id:UUID}/exclude * (Server-authenticated TLS connection)

Get the current host system's exclude list associated to the Cloud Agent {agent_id}. It internally invokes the corresponding API exposed by the CV.

Response JSON object

- `exclude` (list[string]): host system's exclude list.

Example Response:

```
{
   "code": 200,
   "status": "Success",
   "results": {
                "exclude": [ "reg_ex1", "reg_ex2", ... ]
             }
}
```

PUT /v2/agents/{agent_id:UUID}/exclude * (Server-authenticated TLS connection)

Substitute the host system's exclude list associated to the Cloud Agent {agent_id} with the one provided in the request body. To reset the exclude list, provide an empty list in the request body. This API internally invokes the corresponding one exposed by the CV.

Request JSON object

- `e_list_data` (list[string]): the new host system's exclude list.

Example Request:

```
{
   "e_list_data": [ "reg_ex1", "reg_ex2", ... ]
}
```

Response JSON object

- `exclude` (list[string]): the newly updated host system's exclude list.

Example Responses:

```
{
   "code": 200,
   "status": "Success",
   "results":  {
                "exclude": [ "reg_ex1", "reg_ex2", ... ]
             }
}
```

```
{
   "code": 400,
   "status": "Invalid regex:  " + regex_err.msg + ".",
   "results":  {}
}
```

Response code 400 can occur if some of the regular exception provided in the request body is not valid.

GET /v2/agents/{agent_id:UUID}/containers * (Server-authenticated TLS connection)

Get the list of all container identifiers currently registered in Cloud Agent {agent_id}. This API will internally invoke the corresponding one exposed by the CV.

Response JSON object

- `agent_id` (string): Cloud Agent's UUID;
- `cont_ids` (list[string]): list of registered container identifiers.

Example Response:

```
{
   "code": 200,
   "status": "Success",
   "results": {
               "agent_id": "agent_id",
               "cont_ids": [ "contID1", "contID2", ... ]
             }
  }
```

POST /v2/agents/{agent_id:UUID}/containers * (Server-authenticated TLS connection)

Add new containers to those already associated to the Cloud Agent {agent_id}. If the identifier of one of the containers in the request body is already registered in the CV, the request is rejected. This API will internally invoke the corresponding one exposed by the CV.

Request JSON object

Each key in the request body is the ID of a new container to be registered; the JSON object associated to it contains the following fields:

- `a_list_data` (list[string]): it is the allowlist associated to the container; each string inside the list has the format "hash file-path";
- `e_list_data` (list[string]): it is the exclude list associated to the container.

Example Request:

```
{
   "c4e7bbd89a01": {
                    "a_list_data": [
                                     "hash1_sha1    /path/to/file1",
                                     "hash2_sha256  /path/to/file1",
                                     "hash3_sha1    /path/to/file2",
                                     "hash4_sha256  /path/to/file2",
                                       ...
                                   ],
```

```
                    "e_list_data": [ "/var/log/*", "/root/etc/fstab", ... ]
                  },
    "ef9cca80b16d": { ... }
}
```

Response JSON object

The response body contains information about all the containers currently associated to the Cloud Agent {agent_id}, each one represented by the following fields:

- `operational_state` (int): it is the container integrity state, which can be: 0 = START (it is waiting the first integrity check), 1 = TRUST, 2 = UNTRUST;
- `allowlist` (JSON object): it is the container's allowlist;
- `exclude` (list[string]): it is the container's exclude list;
- `fnf` (list[string]) - *optional*: it is the list of file paths not found in the container's allowlist during remote attestation;
- `filehash_err` (list[string]) - *optional*: it is the list of file paths whose digest do not match the ones contained in the container's allowlist.

Example Response:

```
{
    "code": 200,
    "status": "Success",
    "results": {
                "017e5a179cba": {
                                  "operational_state": 2,
                                  "allowlist": {
                                                  "f_path1": ["hash1", "hash2", ...],
                                                  "f_path2": ["hash3", "hash4", ...],
                                                    ...
                                                },
                                  "exclude": [ "reg_ex1", "reg_ex2", ... ],
                                  "fnf": [ "path_fnf1", "path_fnf2", ... ],
                                  "filehash_err": [ "path_1", "path_2", ... ]
                                },
                "8ffae5cb4176": { ... },
                "c4e7bbd89a01": { ... },
                "ef9cca80b16d": { ... }
}

{
    "code": 409,
    "status": "The container " + container_id + " is already registered in agent " +
     agent_id,
    "results": {}
}
```

Response code 409 occur when one of the containers listed in the request body is already registered in the agent. In order to update an already registered container, use PUT request instead.

PUT /v2/agents/{agent_id:UUID}/containers * (Server-authenticated TLS connection)

Substitute the containers currently associated to the Cloud Agent {agent_id} with those provided in the request body. This API will internally invoke the corresponding API of the CV.

Request JSON object

Each key in the request body is the ID of a container to be registered or updated; the JSON object associated to it contains the following fields:

- `a_list_data` (list[string]): it is the allowlist associated to the container; each string inside the list has the format "hash file-path";
- `e_list_data` (list[string]): it is the exclude list associated to the container.

Example Request:

```
{
   "67adbce018bd": {
                      "a_list_data": [
                                        "hash1_sha1    /path/to/file1",
                                        "hash2_sha256  /path/to/file1",
                                        "hash3_sha1    /path/to/file2",
                                        "hash4_sha256  /path/to/file2",
                                           ...
                                      ],
                      "e_list_data": [ "/var/log/*", "/root/etc/fstab", ... ]
                    },
   "39fc5aeb20de": { ... },
     ...
}
```

GET /v2/agents/{agent_id:UUID}/containers/{container_id} *

(Server-authenticated TLS connection)

Retrieve information about the container {container_id} associated to the Cloud Agent {agent_id}. This API will internally invoke the corresponding API of the CV.

Response JSON object

- `container` (JSON object): it contains the information related to the {container_id}.

Example Response:

```
{
   "code": 200,
   "status": "Success",
   "results": {
                "container": {
                                "contID": "5ef8c19baead",
                                "operational_state": 2,
                                "allowlist": {
                                                "f_path1": [ "hash1", "hash2", ... ],
                                                "f_path2": [ "hash3", "hash4", ... ],
                                                  ...
                                              },
                                "exclude": [ "reg_ex1", "reg_ex2", ... ],
                                "fnf": [ "fnf1", "fnf2", ... ],
                                "filehash_err": [ "f_err1", "f_err2", ... ]
                              }
              }
}
```

PUT /v2/agents/{agent_id:UUID}/containers/{container_id} *

(Server-authenticated TLS connection)

Update the allowlist and exclude list of the {container_id} associated to the Cloud Agent {agent_id}; if the {container_id} is not registered, add it to the list of containers associated to {agent_id}. This API will internally invoke the corresponding API of the CV

Request JSON object

- `a_list_data` (list[string]): the allowlist of the {container_id}, each string in the list has the format "digest file-path";
- `e_list_data` (list[string]): the exclude list of the {container_id};

Example Request:

```
{
   "a_list_data": [
                     "hash1_sha1    /path/to/file1",
                     "hash2_sha256  /path/to/file1",
                     "hash3_sha1    /path/to/file2",
                     "hash4_sha256  /path/to/file2",
                       ...
                  ],
   "e_list_data": [ "reg_ex1", "reg_ex2", .... ]
}
```

Response JSON object

Example Response:

```
{
   "code": 201,
   "status": "Created",
   "results": {
                "contID": "c91e447a05cb",
                "operational_state": 0,
                "allowlist": {
                                "/path/to/file1": [ "hash1", "hash2", ... ],
                                "/path/to/file2": [ "hash3", "hash4", ... ],
                                  ...
                              },
                "exclude": [ "reg_ex1", "reg_ex2", ... ]
              }
}
```

If the {container_id} has been added, the response code is `201` "Created", if it has been updated, the response code is `200` "Success".

DELETE /v2/agents/{agent_id:UUID}/containers/{container_ID} *

(Server-authenticated TLS connection)

Remove the {container_id} from the Cloud Agent {agent_id}. This API will invoke the corresponding API exposed by the CV.

GET /v2/agents/{agent_id:UUID}/containers/{container_id}/allowlist *

(Server-authenticated TLS connection)

Get the current allowlist of {container_id}, registered in the CV and associated to the Cloud Agent {agent_id}. This API will invoke the corresponding API of the CV in order to retrieve the required information.

Response JSON object

- `allowlist` (JSON object): the current {container_id}'s allowlist.

Example Response:

```
{
   "code": 200,
   "status": "Success",
   "results": {
                "allowlist": {
                               "/path/to/file1": [ "hash1", "hash2", ... ],
                               "/path/to/file2": [ "hash3", "hash4", ... ],
                                 ...
                             }
              }
}
```

PUT /v2/agents/{agent_id:UUID}/containers/{container_id}/allowlist *

(Server-authenticated TLS connection)

Substitute the allowlist of {container_id}, associated to the Cloud Agent {agent_id}, with the one provided in the request body. This API will invoke the corresponding API of the CV for updating the information in the CV's database.

Request JSON object

- `allowlist` (list[string]): the new allowlist for {container_id}; each string of the list has the format "digest file-path".

Example Request:

```
{
   "a_list_data": [
                    "hash1_sha1    /path/to/file1",
                    "hash2_sha256  /path/to/file1",
                    "hash3_sha1    /path/to/file2",
                    "hash4_sha256  /path/to/file2",
                      ...
                  ]
}
```

PATCH /v2/agents/{agent_id:UUID}/containers/{container_id}/allowlist *

(Server-authenticated TLS connection)

Patch the allowlist of {container_id}, registered in the CV and associated to the Cloud Agent {agent_id}. This API internally invokes the corresponding API of the CV to update the information in the CV's database.

Request JSON object

- `delete_paths` (list[string]) - *optional* : the list of file paths to be removed from the {container_id} allowlist;
- `put_paths` (list[string]) - *optional* : the file paths to be added or updated in the {container_id} allowlist; each string in the list has the format "digest file-path".

Example Request:

```
{
   "delete_paths": [ "/path/to/file1", "/path/to/file2", ... ],
   "put_paths": [
                  "hash1_sha1   /path/to/file3",
                  "hash2_sha256 /path/to/file3",
                     ...
                ]
}
```

Response JSON object

- `allowlist` (JSON object): the newly updated {container_id} allowlist.

Example Responses:

```
{
   "code": 200,
   "status": "Success",
   "results":  {
                 "allowlist": {
                                 "/path/to/file1": [ "hash1", "hash2", ... ],
                                 "/path/tp/file2": [ "hash3", "hash4", ... ],
                                     ...
                               }
               }
}

{
   "code": 400,
   "status": "The request body does not contain the fields \"delete_paths\" or
    \"put_paths\"",
   "results": {}
}
```

Response code `400` can occur if request body is empty or does not contain neither `delete_paths` nor `put_paths`.

GET /v2/agents/{agent_id:UUID}/containers/{container_id}/exclude *

(Server-authenticated TLS connection)

Get the current exclude list of {container_id}, registered in the CV and associated to the Cloud Agent {agent_id}. This API internally invokes the corresponding CV's APIs in order to retrieve the required information.

Response JSON object

- `exclude` (list[string]): the {container_id}'s exclude list.

Example Response:

```
{
   "code": 200,
   "status": "Success",
   "results": {
                "exclude": [ "reg_ex1", "reg_ex2", ... ]
              }
}
```

PUT /v2/agents/{agent_id:UUID}/containers/{container_id}/exclude *

(Server-authenticated TLS connection)

Substitute the exclude list of {container_id}, associated to the Cloud Agent {agent_id}, with the one provided in the request body. This API will invoke the corresponding API of the CV to update the information in the CV's database.

Request JSON object

- `e_list_data` (list[string]): the new exclude list for the {container_id}.

Example Request:

```
{
   "e_list_data": [ "reg_ex1", "reg_ex2", ... ]
}
```

Response JSON object

- `exclude` (list[string]): the newly updated exclude list of {container_id}.

Example Responses:

```
{
   "code": 200,
   "status": "Success",
   "results": {
                "exclude": [ "reg_ex1", "reg_ex2", ... ]
              }
}

{
   "code": 400,
   "status": "Invalid regex:  " + regex_err.msg + ".",
   "results": {}
}
```

Response code `400` occurs when one of the regular expressions in the request body is not valid, the request body is empty or does not contain the `e_list_data` field.

Appendix E

Programmer's manual: Whitelists Web Service REST APIs

This appendix describes the REST APIs exposed by the Whitelists Web Service, a new subcomponent of the TM developed with the thesis work. The Whitelists Web Service accepts only JSON format and the responses provided by APIs are, for the most part, JSON objects with the following format:

- `code` (int): the HTTP status code;
- `status` (string): textual explanation of the response status;
- `results` (JSON object): it contains specific data of the response and its format depends on the invoked API.

The only APIs with a different response format are `GET /containers/{container_id}` and `GET /hosts/{host_id}`, which return a string containing the whitelist of a container or a physical host respectively, in the format expected by the Keylime framework.

GET /hosts/{host_id} (Unencrypted connection)

Retrieve the whitelist for the host {host_id}.

Response

The response is the {host_id}'s whitelist expressed in the format that the Keylime framework expects; each line of the string contains a "digest file-path" pair:

```
"7ca9503922742b2[...]1891c6d0b179f9bf7bcfd3d  /usr/bin/dockerd\n
 8d9ae560cc518a6[...]afc4f5d2134ceec77a87d2f  /usr/bin/docker-init\n
 640bfa79de72a16[...]f6d3be02f0c80561b8e9784  /usr/bin/docker-proxy\n
 119eb7f82994502[...]77adf198d6ae170b044146f  /usr/bin/getent\n
 93e226f3876bb17[...]ca82941f4957d69f8e1a9bd  /usr/bin/cut\n
 331461536894ebf[...]dde380adfdfc4edd8a258d2  /usr/bin/md5sum\n
 d10d2c03eb3aa9c[...]c7e3fced895329afc914d1d  /usr/bin/head\n
 ..."
```

PUT /hosts/{host_id} (Unencrypted connection)

Create or update the whitelist for {host_id}.

Request JSON object

- `architecture` (string): host system's architecture, for example "amd64", "i386", "arm64";
- `hash_algorithms` (string): a comma-separated list of hash algorithms that have to be contained in the whitelist; the hash algorithms currently supported are: "sha1", "sha256", "sha384", "sha512"; not supported algorithms will be ignored;
- `software_config` (JSON object) -*optional*: if `package_list` is specified, this field is not admitted; it specifies the software configuration of the host system; each object's key represents a system's software and should be declared in the `whitelists-sources-names.txt` configuration file; the JSON objects describing the software components contain the following fields:
  - `origin` (string): the operative system's name, for example "Ubuntu";
  - `distributions` (string): a comma separated list of distributions, for example "focal, focal-security";
  - `components` (string): a comma separated list of distributions' components, for example "main, universe";
- `packages_list` (string) -*optional*: if `software_config` is specified, this field is not admitted; it is a list of rows containing the packages installed on the host system; each row has the format "package-name package-version architecture\n";
- `known_digests` (list[JSON object]) -*optional*: custom digests, related to proprietary software, that have to be added to the host system's whitelist; each `known_digest` is a JSON object containing the following fields;
  - `hash` (string): custom digest;
  - `path` (string): file path to which the digest refers.

Example Request 1:

```
{
   "architecture": "amd64",
   "hash_algorithms": "sha1,sha256",
   "software_config": {
                        "Operative System": {
                                              "origin": "Ubuntu",
                                              "distributions": "focal, focal-security,
                                                               focal-updates",
                                              "components": "main, restricted,
                                                             universe, multiverse"
                                            },
                        "Docker": {
                                    "origin": "Ubuntu",
                                    "distributions": "focal",
                                    "components": "stable"
                                  },
                          ...
                        },
   "known_digests": [
                       {
                         "hash": "56ad914cc[...]3401e",
                         "path": "/path/to/file"
                       },
                         ...
                     ]
}
```

Example Request 2:

```
{
   "architecture": "amd64",
   "hash_algorithms": "sha512",
   "packages_list": "accountsservice 0.6.55-0ubuntu12 20.04.4 amd64\n
                     adduser 3.118ubuntu2 all\n
                     alsa-topology-conf 1.2.2-1 all\n
                     alsa-ucm-conf 1.2.2-1ubuntu0.8 all\n
                     amd64-microcode 3.20191218.1ubuntu1 amd64\n
                     apparmor 2.13.3-7ubuntu5.1 amd64\n
                     apport 2.20.11-0ubuntu27.18 all\n
                     ...",
   "known_digests": [
                       {
                         "hash": "56ad914cc[...]3401e",
                         "path": "/path/to/file"
                       },
                        ...
                      ]
}
```

Response JSON object

- `host_id` (string): the host's identifier;
- `architecture` (string): the host's architecture;
- `hash_algorithms` (list[string]): the hash algorithms with which the whitelist was built;
- `software_config` (JSON object): the software configuration specified in the request body;
- `packages_list` (string): the installed packages specified in the request body;
- `whitelist` (string): the created or updated whitelist for the {host_id}, available through GET /hosts/{host_id}.

Example Response 1:

```
{
   "code": 201,
   "status": "Created",
   "results": {
               "host_id": "D432FBB3-D2F1-4A97-9EF7-75BD81C00000",
               "architecture": "amd64",
               "hash_algorithms": ["sha1", "sha256"],
               "software_config": {
                                    "Operative System": {
                                                          "origin": "Ubuntu",
                                                          "distributions": "focal,
                                                                      focal-security,
                                                                      focal-updates",
                                                          "components": "main,
                                                                        restricted,
                                                                        universe,
                                                                        multiverse"
                                                        },
                                     "Docker": {
                                                 "origin": "Ubuntu",
                                                 "distributions": "focal",
                                                 "components": "stable"
                                               },
                                      ...
                                   },
               "whitelist": "137dfa[...sha1...]a3563d   /path/to/file1\n
```

```
                                acbb3[...sha256...]99ea5   /path/to/file1\n
                                22ea1f[...sha1...]8abd1f  /path/to/file2\n
                                1d76a[...sha256...]9ecae   /path/to/file2\n
                                ..."
                }
    }
```

Example Response 2:

```
{
    "code": 200,
    "status": "Success",
    "results": {
                "host_id": "FDFSE84-KSRFE900-39DJ74",
                "architecture": "amd64",
                "hash_algorithms": ["sha1", "sha256"],
                "packages_list": "accountsservice 0.6.55-0ubuntu12 20.04.4 amd64\n
                                  adduser 3.118ubuntu2 all\n
                                  alsa-topology-conf 1.2.2-1 all\n
                                  alsa-ucm-conf 1.2.2-1ubuntu0.8 all\n
                                  amd64-microcode 3.20191218.1ubuntu1 amd64\n
                                  apparmor 2.13.3-7ubuntu5.1 amd64\n
                                  apport 2.20.11-0ubuntu27.18 all\n
                                  ...",
                "whitelist": "137dfa[...sha1...]a3563d  /path/to/file1\n
                              acbb3[...sha256...]99ea5  /path/to/file1\n
                              22ea1f[...sha1...]8abd1f  /path/to/file2\n
                              1d76a[...sha256...]9ecae  /path/to/file2\n
                              ..."
                }
    }
```

The response code is `201` when the {host_id}'s whitelist has been created, it is `200` when the whitelist has been updated.

POST /hosts/{host_id}/known_digest (Unencrypted connection)

Add to the {host_id}'s whitelist the custom digest specified in the request body. If the whitelist resource has not yet been created, the request is rejected.

Request JSON object

- `hash` (string): custom digest;
- `path` (string): file path to which the digest refers.

Example Request:

```
{
    "hash": "34edd[...]cd91a",
    "path": "/path/to/file"
}
```

GET /packages/{package_name} (Unencrypted connection)

Retrieve all the information for the {package_name}.

Response JSON object

- `package_name` (string): the required {package_name};
- `origin` (string): the operative system on which the package can be installed;
- `software_name` (string): the "symbolic" name of the software to which the package belongs, as defined in the configuration file `whitelists-sources-names.txt`;
- `distribution` (string): the operative system's distribution on which the package can be installed;
- `component` (string): the component to which the package belongs;
- `architectures` (JSON object): the architectures, present in the local database, on which the package can be installed; each architecture is represented by a JSON obect containing the package versions downloaded in the local database; each package version describes all the features of the package, including its whitelist.

Example Response:


```
{
  "code": 200,
  "status": "Success",
  "results": {
         "package_name": "containerd.io",
         "origin": "Ubuntu",
         "software_name": "Docker",
         "distribution": "focal",
         "component": "stable",
         "architectures": {
               "amd64": {
                    "1.2.13-2": {
                         "Package": "containerd.io",
                         "Architecture": "amd64",
                         "Version": "1.2.13-2",
                         "Priority": "optional",
                         "Section": "devel",
                         "Maintainer": "Containerd team <help@containerd.io>",
                         "Installed-Size": "96972",
                         "Provides": "containerd, runc",
                         "Depends": "libc6 (>= 2.14), libseccomp2 (>= 2.4.1)",
                         "Conflicts": "containerd, runc",
                         "Replaces": "containerd, runc",
                         "Filename": "dists/focal/pool/stable/amd64
                                     /containerd.io_1.2.13-2_amd64.deb",
                         "Size": "21420058",
                         "MD5sum": "82a480e21a52caba100623cf534532e2",
                         "SHA1": "18bce1e3a4a1cafeb5ef8eafa6766225be195125",
                         "SHA256": "96ad73534f896e1d[...]33de8183b571932939a4f740",
                         "SHA512": "ab86f4e14362b2ee[...]5a5275403c70b293da18ea86c23",
                         "Homepage": "https://containerd.io",
                         "Description": "An open and reliable container runtime",
                         "File_uri": "https://download.docker.com/linux/ubuntu
                              [...]/stable/amd64/containerd.io_1.2.13-2_amd64.deb",
                         "Is_security_update": false,
                         "Whitelist": {
                              "/etc/containerd/config.toml": {
                                   "sha1": "b12dedffdab63138a0a1484dde9ed95c283aca23",
                                   "sha256": "d66bdfd27e0817dd0c2702035[...]30096f46c",
                                   "sha384": "e9b46c91a82fa4602e1cc[...]7d268e73d61f9",
                                   "sha512": "3ccef5ce455db823d4[...]60a0b3898c05b1640"
                               },
                               ...
                         }
                    },
                    "1.3.7-1": { ... },
```

```
                ...
            },
            "i386": { ... }
        }
    }
}
```

PUT /packages/{package_name} (Unencrypted connection)

Update or download the {package_name} for the version and architecture specified in request body.

Request JSON object

- `version` (string): the requested version of {package_name};
- `architecture` (string): the requested architecture.

Example Request for package "bash":

```
{
   "version": "5.0-6ubuntu1.1",
   "architecture": "amd64"
}
```

Response JSON object

It contains all the information related to the downloaded or updated package, including its whitelist.

Example Response:

```
{
   "code": 200,
   "status": "Success",
   "results": {
         "Package": "bash",
         "Architecture": "amd64",
         "Version": "5.0-6ubuntu1.1",
         "Multi-Arch": "foreign",
         "Priority": "required",
         "Essential": "yes",
         "Section": "shells",
         "Origin": "Ubuntu",
         "Maintainer": "Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>",
         "Original-Maintainer": "Matthias Klose <doko@debian.org>",
         "Bugs": "https://bugs.launchpad.net/ubuntu/+filebug",
         "Installed-Size": "1656",
         "Pre-Depends": "libc6 (>= 2.15), libtinfo6 (>= 6)",
         "Depends": "base-files (>= 2.1.12), debianutils (>= 2.15)",
         "Recommends": "bash-completion (>= 20060301-0)",
         "Suggests": "bash-doc",
         "Conflicts": "bash-completion (<< 20060301-0)",
         "Replaces": "bash-completion (<< 20060301-0), bash-doc (<= 2.05-1)",
         "Filename": "pool/main/b/bash/bash_5.0-6ubuntu1.1_amd64.deb",
         "Size": "638312",
         "MD5sum": "a8e68bda652adfc5173d683819426ee8",
         "SHA1": "5d302f347e31fea7a7649188acf11841e68116b6",
         "SHA256": "235fc2622fbc61f7f5548f2d80abda3[...]cba8e3442de7a69f3",
         "SHA512": "87bdb291260da0838e5c8bd95a215[...]9eed05b3d897f8d8f090489",
         "Homepage": "http://tiswww.case.edu/php/chet/bash/bashtop.html",
         "Description": "GNU Bourne Again SHell",
```

```
            "Task": "minimal",
            "Description-md5": "3522aa7b4374048d6450e348a5bb45d9",
            "File_uri": "http://archive.ubuntu.com/[...]/bash_5.0-6ubuntu1.1_amd64.deb",
            "Whitelist": {
                        "/bin/bash": {
                                "sha1": "41ba1bd49cb22466e422098d[...]ef9529e",
                                "sha256": "04a484f27a4b485b284519[...]9fb5f2eea9",
                                "sha384": "2c1c5d6474235f94bc500f4[...]a5ab60cb2d7",
                                "sha512": "b9bff09b39fbaa8db91d081[...]e8680f33306be1"
                         },
                        "/etc/bash.bashrc": {
                                "sha1": "cbd89fb1fa310fc4bc46866081[...]922cd2",
                                "sha256": "29128d49b590338131373e[...]17ac9a25",
                                "sha384": "907f5eb888047aa8197253[...]b19b7f3627",
                                "sha512": "747f3d9ba51c27852c4f2eb[...]8b88806cff95"
                         },
                          ...
                }
    }
}
```


If {package_name}, for the requested version and architecture, is not found in the remote repositories configured in the file `whitelists-packages-sources.list`, the response code is 404 "Not Found".

GET /images/{image_id} (Unencrypted connection)

Retrieve information related to container image {image_id}.

Response JSON object

- `image_id` (string): the identifier with which the image has been registered in the database;
- `rootfs` (string): the rootfs of the image;
- `layers` (JSON object): it contains the files divided by the layers of which the image is made up;
- `whitelist` (JSON object): it contains the whitelist of the image; if the same file appears on several levels, the digest considered is that of the highest level.

Example Response:


```
{
    "code": 200,
    "status": "Success",
    "results": {
                "image_id": "Keylime",
                "rootfs": "quay.io/fedora/fedora:34-x86_64",
                "layers": {
                            "0": {
                                   "/path/to/file1": {
                                                "sha1": "23dda[...]3cc",
                                                "sha256": "c710dd2[...]fcb3",
                                                "sha384": "445e[...]74fce",
                                                "sha512": "9416c[...]d131"
                                   },
                                   "/path/to/file2": { ... },
                                   ...
                             },
                            "1": {
```

```
                                "/path/to/file1": {...},
                                "/path/to/file3": {...},
                                  ...
                            },
                            "2": {
                                "/path/to/file2": {...},
                                "/path/to/file4": {...},
                                  ...
                            }
                    },
                    "whitelist": {
                                "/path/to/file1": {
                                                    "sha1": "55dea[...]abc1",
                                                    "sha256": "47aa2[...]01d2a",
                                                    "sha384": "6adcf4[...]33ed",
                                                    "sha512": "aac1e[...]4effb"
                                                    },
                                "/path/to/file2": {...},
                                "/path/to/file3": {...},
                                "/path/to/file4": {...},
                                 ...
                            }
            }
}
```

PUT /images/{image_id} (Unencrypted connection)

Create or update the whitelist for the container image specified in the request body and identified as {image_id}.

Request JSON object The two fields that can be specified in the request body are mutually exclusive but at least one of them is required:

- `image_to_pull` (string) -*optional*: it specifies the name of the image to be pulled from the configured Docker repository;
- `image_to_build` (string) -*optional*: it specifies the remote URI or the local path of the image to be build.

Example Request 1:

```
{
   "image_to_pull": "ubuntu"
}
```

Example Request 2:

```
{
   "image_to_build": "https://github.com/keylime/keylime.git#:docker/ci"
}
```

The body of the response is the same as shown for the `GET /images/{image_id}` API; the image resource has been created when the response code is `201`, it has been updated when the response code is `200`.

GET /containers/{container_id} (Unencrypted connection)

Retrieve {container_id}'s whitelsit.

Response

The response is the {container_id}'s whitelist expressed in the format that the Keylime framework expects; each line of the string contains a "digest file-path" pair:

```
"f1a9f328e6a0856[...]2ffc89e40baf23025a68829   /usr/local/sbin/unminimize\n
 3222cd943a02ac5[...]f6a93ce8cc11fa7c2337fc2   /usr/sbin/grpck\n
 1a07adb8bc0cf9f[...]415c400fb7dede623dd4fb9   /usr/sbin/rmt-tar\n
 2ca2667bb07c833[...]d7f5ef41ca5ad5c0ca5b6fb   /usr/sbin/pwck\n
 533ccdf21c79a22[...]159cb1dec20811fbd723456   /usr/sbin/blkzone\n
 adf87d9242cab10[...]a34144703a312c12aa14f9f   /usr/sbin/ldattach\n
 d20ed1c7d1e8fed[...]2f9839f4aaa51eed4dff0da   /usr/sbin/switch_root\n
 5298aac043b5b45[...]11a837c7aba5e904e788971   /usr/sbin/groupmems\n
 dcfc43a37bf2680[...]8587435e290b2c1e0e856de   /usr/sbin/isosize\n
 982cca7d6a9afe0[...]88d253f6464bfc0a19730674  /usr/sbin/pam_getenv\n
 ..."
```

PUT /containers/{container_id} (Unencrypted connection)

Create or update the {container_id}'s whitelist.

Request JSON object

- `image_id` (string): it is the image identifier used for creating the image resource through the `PUT /images/{image_id}` API;
- `hash_algorithms` (string): it is a comma separated list of hash algorithms to be used for the creation of the whitelist; if a specified hash algorithm is not supported, it will be ignored; the hash algorithm currently supported are: sha1, sha256, sha384, sha512.

Example Request:

```
{
   "image_id": "Keylime",
   "hash_algorithms": "sha256"
}
```

Response JSON object

- `container_id` (string): the {container_id} specified in the URI;
- `image_id` (string): the `image_id` specified in the request body;
- `hash_algorithms` (list[string]): the list of hash algorithms of which the whitelist is composed;
- `whitelist` (string): the {container_id}'s whitelist.

Example Response to the `PUT /containers/37aba78dcc2b` request:

```
{
   "code": 200,
   "status": "Success",
   "results": {
         "container_id": "37aba78dcc2b",
         "image_id": "ubuntu",
         "hash_algorithms": [ "sha256" ],
         "whitelist": "f1a9f328e6a0856[...]3025a68829 /usr/local/sbin/unminimize\n
                       3222cd943a02ac5[...]a7c2337fc2 /usr/sbin/grpck\n
                       1a07adb8bc0cf9fb[...]e623dd4fb9 /usr/sbin/rmt-tar\n
                       2ca2667bb07c833[...]d5c0ca5b6fb /usr/sbin/pwck\n
```

```
                    533ccdf21c79a224[...]11fbd723456 /usr/sbin/blkzone\n
                    adf87d9242cab10f[...]12c12aa14f9f /usr/sbin/ldattach\n
                    d20ed1c7d1e8fed6[...]51eed4dff0da /usr/sbin/switch_root\n
                    5298aac043b5b450[...]e904e788971 /usr/sbin/groupmems\n
                    dcfc43a37bf2680e7[...]2c1e0e856de /usr/sbin/isosize\n
                    982cca7d6a9afe07d[...]c0a19730674 /usr/sbin/pam_getenv\n
                    2d66f8b37035b68bf[...]397969120e3 /usr/sbin/fdisk\n
                    ..."
  }
}
```

Appendix F

Programmer's manual: Trust Monitor REST APIs

This appendix describes the REST APIs exposed by the TM. The changes introduced to the APIs with the thesis work are highlighted in the text with the character "*".

GET /register_node/ (Server-authenticated TLS connection)

Get the list of all physical hosts registered with the TM.

Response list of JSON objects

- `id` (int): host identifier in the TM's database;
- `hostName` (string): host's UUID;
- `address` (string): host's IP address;
- `pcr0` (string): it is the content of TPM's PCR 0 register; it has a significant value only for hosts registered with OAT attestation driver;
- `distribution` (string): it defines the host's OS; for Keylime attestation driver, it is the dump of a JSON object describing host's software and hardware configuration, to be passed to the Whitelists Web Service for creating the host whitelist;
- `analysisType` (string): it specifies the analysis type to be applied for attesting the node;
- `driver` (string): it is the attestation driver to be used for the node;
- `containers*` (string): it specifies the container ids registered with the node, with the corresponding image identifiers;
- `allowUnknownContainers*` (int): it specifies whether unknown containers are allowed on the host system.

Example Response:

```
[
   {
      "id": 1,
      "hostName": "host1",
      "address": "192.168.1.50",
      "pcr0": "",
```

```
    "distribution": "{
                      \"architecture\":\"amd64\",
                      \"hash_algorithms\":\"sha256\",
                      \"packages_list\":\"acl 2.2.53-6 amd64\n
                                          adduser 3.118ubuntu2 all\n
                                          ..."
                    }",
    "analysisType": "load-time+cont-check,l_req=l4_ima_all_ok|==,cont-list=",
    "driver": "Keylime",
    "containers": "f47a5af906cf ubuntu\n
                   39bf9c4591df fedora",
    "allowUnknownContainers": 1
  }
]
```

POST /register_node/ (Server-authenticated TLS connection)

Register a new node inside the TM.

Request JSON object

- `hostName` (string): host's UUID;
- `address` (string): host's IP address;
- `distribution` (string): the host's OS; if the attestation driver is "Keylime", it is the host software and hardware configuration to be passed to the Whitelists Web Service to create the whitelist;
- `driver` (string): it is the attestation driver to be used for verifying the integrity state of the node;
- `pcr0` (string) -*optional*: it is the TPM PCR 0 value; this field is used only by OAT attestation driver;
- `containers`* (string) -*optional*: it specifies the list of containers running on the host; each line of the string indicates a container with the format "container-id image-id", where "image-id" is the identifier with which the container's image is registered in the Whitelists Web Service database;
- `allowUnknownContainers`* (int) -*optional*: it specifies if unknown containers are admitted on the host.

Example Request:

```
{
  "hostName": "host1",
  "address": "192.168.1.50",
  "distribution": "{
                    \"architecture\":\"amd64\",
                    \"hash_algorithms\":\"sha256\",
                    \"packages_list\":\"acl 2.2.53-6 amd64\n
                                        adduser 3.118ubuntu2 all\n
                                        ..."
                  }",
  "driver": "Keylime",
  "containers": "f47a5af906cf ubuntu\n
                 39bf9c4591df fedora",
  "allowUnknownContainers": 1
}
```

Response JSON object

If an host with IP `address` specified in the request body already exists in the TM's database, the response code is `200` and no operation is performed. If the IP `address` is not present in the database and all the fields in the request body have correct format, the response code is `201` "Created" and the response body has the following format (where `id` is the host's identifier in the TM's database):

```
{
   "id": 1,
   "hostName": "host1",
   "address": "192.168.1.50",
   "pcr0": "",
   "distribution": "{
                     \"architecture\":\"amd64\",
                     \"hash_algorithms\":\"sha256\",
                     \"packages_list\":\"acl 2.2.53-6 amd64\n
                                         adduser 3.118ubuntu2 all\n
                                         ..."
                  }"
   "analysisType": "load-time+cont-check,l_req=l4_ima_all_ok|==,cont-list=",
   "driver": "Keylime",
   "containers":"f47a5af906cf ubuntu\n
                 39bf9c4591df fedora",
   "allowUnknownContainers": 1
}
```

DELETE /register_node/ (Server-authenticated TLS connection)

Remove the host specified in the request body from the TM.

Request JSON object

- `hostName` (string): host's UUID.

Example Request:

```
{
   "hostName": "host1"
}
```

Response JSON object

If a host with name specified in the request body does not exist, the response code is `403` "Forbidden"; otherwise, if the host exists and the request body is correct, the response code is `200` "OK" and the response body has the following format:

```
{
    "Host host1": "removed"
}
```

GET /status/ (Server-authenticated TLS connection)

Get the status of all subcomponents inside the TM architecture.

Response list of JSON objects

The response is a list of three JSON objects for describing the status of attestation drivers, connectors and databases. The status of each subcomponent is defined by two fields:

- `active` (bool): it specifies if the subcomponent is running;
- `configuration` (bool): it specifies if the subcomponent is correctly configured.

Example Response:


```
[
   {
      "drivers": [
                    {
                       "OAT": {
                                "active": false,
                                "configuration": false
                              }
                    },
                    {
                       "OpenCIT": {
                                    "active": false,
                                    "configuration": false
                                  }
                    },
                    {
                       "HPESwitch": {
                                      "active": false,
                                      "configuration": true
                                    }
                    },
                    {
                       "Keylime": {
                                    "active": true,
                                    "configuration": true
                                  }
                       }
                 ]
    },
    {
       "connectors": [
                        {
                           "DARE": {
                                     "active": false,
                                     "configuration": true
                                   }
                        },
                        {
                           "Dashboard": {
                                          "active": true,
                                          "configuration": true
                                        }
                        },
                        {
                           "VNSFO": {
                                      "active": false,
                                      "configuration": true
                                    }
                        },
                        {
                           "VIM-EMU": {
                                        "active": false,
                                        "configuration": true
                                      }
                        },
                        {
                            "vNSF Store": {
                                           "active": true,
                                           "configuration": true
```

```
                                        }
                    }
                ]
    },
    {
       "databases": [
                    {
                       "whitelist-db": {
                                         "active": false,
                                         "configuration": false
                                        }
                    },
                    {
                       "known-digests": {
                                         "active": true,
                                         "configuration": true
                                        }
                    }
                ]
    }
]
```

POST /attest_node/ (Server-authenticated TLS connection)

Check the integrity status of the nodes specified in the request body.

Request JSON object

- `node_list` (list[JSON object]): it specifies the nodes to be attested; each node of the list is described by the following fields:
  - `node` (string): the name of the host to be attested;
  - `vnfs` (list[string]) -*optional*: the list of containers, running on the node, to be attested.

Example Request:

```
{
   "node_list": [
                  {
                    "node": "host1",
                    "vnfs": [ "39bf9c4591df", "821b05ccef64", ... ]
                  },
                  {
                    "node": "host2"
                  }
                ]
}
```

Response JSON object

- `hosts` (list[JSON object]): it contains the list of computed nodes with the details of the attestation process; in the case of the driver "Keylime", the `extra_info` field does not yet have significant values (this is left to future work);
- `sdn` (list[JSON object]): it is the list of attested SDN nodes;
- `trust` (bool): it is the trust level of the entire NFV / cloud platform;
- `vtime` (string): the time in which the attestation process was performed.

Example Response:

```
{
   "hosts": [
               {
                  "node": "host1",
                  "status": 0,
                  "time": "2021-07-18 23:46:25.765066 +0000 UTC",
                  "remediation": {
                                    "terminate": true,
                                    "isolate": true
                                  },
                  "vnsfs": [
                              {
                                "container": "39bf9c4591df",
                                "vnsfr_id": "",
                                "vnsfd_id": "",
                                "remediation": {
                                                 "terminate": false,
                                                 "isolate": false
                                               },
                                "trust": true,
                                "ns_id": ""
                              },
                                ...
                            ],
                  "trust": false,
                  "driver": "Keylime",
                  "extra_info": {
                                    "n_digests_valid": 0,
                                    "n_packages_valid": 0,
                                    "list_digests_fake_lib": null,
                                    "n_packages_not_security": 0,
                                    "n_packages_unknown": 0,
                                    "n_packages_security": 0,
                                    "list_digests_not_found": null,
                                    "n_digests_not_found": 0,
                                    "n_digests_fake_lib": 0
                                 }
               },
               {
                  "node": "host2",
                   ...
               }
            ],
   "sdn": [],
   "trust": false,
   "vtime": "2021-07-18 23:46:25.768358 +0000 UTC"
}
```


GET /nfvi_attestation_info/ (Server-authenticated TLS connection)

Attest all nodes registered with TM. This API does not receive any parameter and the response body has the format shown for `POST /attest_node/` API.

GET /nfvi_pop_attestation_info/ (Server-authenticated TLS connection)

Attest one node registered with the TM.

Query parameter

- `node_id`: the name of the node to be attested.

Example Request:

```
https://trust_monitor/nfvi_pop_attestation_info/?node_id=host1
```

The response body has the same format shown for `POST /attest_node/` API but the `hosts` list contains only the requested node.

GET /known_digests/ (Server-authenticated TLS connection)

Retrieve the list of all known digest registered in the TM's database.

Response list of JSON objects

Example Response:

```
[
   {
      "id": 1,
      "pathFile": "/usr/bin/bash",
      "digest": "04a484f27a4b485b28451923605d9b528453d6c0"
   },
   {
      "id": 2,
      "pathFile": "/usr/bin/find",
      "digest": "92a2bade19a90a1bd81e4d2c2de646ddf971aba9"
   },
   ...
]
```

POST /known_digests/ (Server-authenticated TLS connection)

Add a known digest in the TM's database.

Request JSON object

- `pathFile` (string): the file path on the compute node;
- `digest` (string): the SHA-1 digest of the file (the API has not yet been extended to support other hash algorithms).

Example Request:

```
{
   "pathFile": "/usr/bin/bash",
   "digest": "04a484f27a4b485b28451923605d9b528453d6c0"
}
```

Response JSON object

- `id` (int): the id of the known digest in the TM's database;
- `pathFile` (string): the added file path;
- `digest` (string): the added SHA-1 digest.

Example Response:

```
{
   "id": 1,
   "pathFile": "/usr/bin/bash",
   "digest": "04a484f27a4b485b28451923605d9b528453d6c0"
}
```

DELETE /known_digests/ (Server-authenticated TLS connection)

Delete a known digest from the TM's database.

Request JSON object

- `digest` (string): the SHA-1 known digest to be removed from the database.

Example Request:

```
{
   "digest": "92a2bade19a90a1bd81e4d2c2de646ddf971aba9"
}
```

Response JSON object

If the digest is found in the database, the response code is `200` "OK" and the digest is correctly removed from the database:

```
{
    "Digest 92a2bade19a90a1bd81e4d2c2de646ddf971aba9": "removed"
}
```

Otherwise, the response code is `403` "Forbidden" and the response body has the following format:

```
{
    "Digest 14f2bade19a90a1bd81e4d2c2de646fdf951aba9": "not found in db"
}
```

POST /audit/ (Server-authenticated TLS connection)

Retrieve historical attestation information for the node specified in the request body. This API internally gets attestation audit from DARE component, since audit functionality is not implemented inside the TM currently.

Request JSON object

- `node_id` (string): the host name; if the request contains only this parameter, the response will contain only the last attestation;
- `from_date` (string) -*optional*: the start date of the time interval to which the required attestation information belongs;
- `to_date` (string) -*optional*: the end date of the time interval to which the required attestation information belongs.

Dates are specified as strings with the format: "%Y-%m-%d %H:%M:%S.%f".
Example Request:

```
{
   "node_id": "host1",
   "from_date": "2021-09-01 00:00:00.0",
   "to_date": "2021-09-02 00:00:00.0"
}
```