POLITECNICO DI TORINO

Master degree course in Ingegneria Informatica

Master Degree Thesis

Privacy-preserving Remote Attestation of pods in Kubernetes

Supervisor
Prof. Antonio Lioy
Ing. Lorenzo Ferro

Candidate
Stefano Caradonna

July 2025

To my family, and to my wise example who taught me to believe in myself.

Summary

The advent of cloud computing has led to a paradigm shift in application and data management, offering greater flexibility, availability and cost-efficiency. Unlike traditional on-premises environments, wherein data and computations are executed locally, cloud-based systems abstract both storage and processing, thereby enabling dynamic resource allocation. The contemporary tendency towards the adoption of fully virtualised environments, wherein multiple tenants' workloads are shared across a common infrastructure, is indicative of this transition. While this model enhances efficiency and reduces operational costs, it also introduces new security challenges, particularly in ensuring the integrity of cloud-based workloads and protecting sensitive data from unauthorised access. In this context, Remote Attestation functions as a security mechanism that helps address the risks associated with shared cloud infrastructures. By analysing system measurements, it verifies that running applications have not been tampered with, thereby providing assurance about the integrity and trustworthiness of cloud-based workloads. These measurements, recorded in Measurement Logs (ML) by the Integrity Measurement Architecture (IMA) in Linux, provide a snapshot of the system's state, ensuring that applications and their dependencies remain unaltered. However, in multi-tenant cloud environments, where multiple verifiers evaluate system integrity, these logs can inadvertently reveal details about unrelated applications, potentially leading to data leaks. The complexity of remote attestation increases in large-scale deployments, where multiple independent verifiers operate without full visibility into the workloads they are assessing, making it difficult to enforce fine-grained access controls. This thesis proposes a solution to the aforementioned challenges by enhancing the traditional attestation framework with privacy-preserving capabilities. The present study proposes a novel IMA template, whose implementation ensures unpredictable measurements and enforces discretion when verifiers access restricted logs. It is evidenced that such restrictions enable the retrieval of data pertinent to the verifier's attestation tasks while ensuring the secrecy of all other system details. A master node will act as a proxy, managing the access control aspect. The initiative adheres to the standards set forth by the Trusted Computing Group (TCG), leveraging the capabilities of the Trusted Platform Module (TPM) 2.0, the IMA, and a widely adopted remote attestation framework. As demonstrated by performance tests conducted in the laboratory, the proposed solution effectively safeguards privacy, mitigates the risk of unauthorized data exposure, and ensures secure pod attestation while concealing cluster details in cloud-native environments. The results further confirm that this approach is highly scalable. It addresses the limitations of existing methods and provides a robust framework for privacy-aware workload verification in Kubernetes-based deployments.

Contents

1 Introduction 9

2 Trusted Computing and Trusted Platform Module TPM 12

2.1 Trusted Computing Group 12

2.2 Trusted Computing Base 12

2.3 Root of Trust 13

2.4 Trusted Platform Building Blocks 14

2.5 Boot Types 14

2.6 TPM 2.0 15

2.6.1 Architecture 15

2.6.2 Main features 15

2.6.3 Securely storing data 16

2.6.4 Implementations 16

2.6.5 TPM Platform Configuration Register PCR 17

2.6.6 Secure Identity 17

2.7 Remote Attestation (RA) 18

2.7.1 Execution flow 19

3 Integrity Measurement Architecture (IMA) 20

3.1 Overview 20

3.2 Design 21

3.2.1 IMA-Measurement 21

3.3 Policies 21

3.4 IMA Templates 22

3.5 ML Integrity Verification and Entries Validation 22

4 Kubernetes (K8s) 25

4.1 Docker container 25

4.2 Overview 26

4.3 State-oriented approach 26

4.4 Architecture 27

4.5 Workload Resources 29

4.5.1 Deployment 29

4.6 Custom Resource 29

5 Keylime Framework 30

5.1 Overview . . . 30

5.2 Design . . . 31

5.3 Main Features . . . 33

5.4 Operational Phases . . . 34

5.4.1 Physical Node Registration Protocol . . . 34

5.4.2 Three Party Bootstrap Key Derivation Protocol . . . 34

5.4.3 Continuous Remote Attestation . . . 36

6 Related works 38

6.1 Container-IMA . . . 38

6.1.1 Architecture . . . 39

6.1.2 New components . . . 40

6.1.3 Measurement Mechanism . . . 41

6.1.4 Attestation Mechanism . . . 43

6.1.5 Consideration . . . 44

6.2 Docker Integrity Verification Engin (DIVE) . . . 44

6.2.1 Consideration . . . 45

6.3 Docker Container Attestation . . . 45

6.4 Kubernetes Pod Attestation . . . 46

6.4.1 Consideration . . . 48

7 Proposed solution - Design 49

7.1 Problems addressed . . . 49

7.2 Proposed Solution . . . 50

7.3 Measurement Log Analysis . . . 50

7.4 Architecture of the Proposed Solution . . . 52

7.5 Registration Process . . . 52

7.5.1 Worker Node . . . 53

7.5.2 Master Node . . . 53

7.5.3 Master Node Capabilities . . . 53

7.5.4 Verifier Registration . . . 54

7.5.5 Verifier Authorisation . . . 55

7.6 Verification Process . . . 55

7.7 Privacy and guessing analysis . . . 58

8 Proposed solution - Implementation 59

8.1 IMA Template and Kernel modification . . . 59

8.2 Keylime modification . . . 60

8.2.1 Tenant modification . . . 62

8.2.2 Agent modification . . . 62

8.2.3 Registrar modification . . . 63

8.2.4 Verifier modification . . . 63

8.3 Kubernetes Capabilities integration . . . 64

8.3.1 Proxy Feature . . . 64

8.3.2 Verifier Authorisation . . . 65

8.3.3 State Aggregation Feature . . . 66

9 Tests 67

9.1 Testbed . . . 67

9.2 Functional Tests . . . 67

9.2.1 Test with a correct Verifier permission . . . 69

9.2.2 Attestation test with an unrecognised Verifier . . . 70

9.2.3 Attestation test with an unrecognised pod for a given Verifier . . . 70

9.3 Comparison with standard Keylime . . . 70

9.4 Performance Tests . . . 71

10 Conclusions and future works 74

Bibliography 76

A User Manual 78

A.1 Attester machine . . . 78

A.1.1 Patching the Linux kernel . . . 78

A.1.2 Preparing the kernel before compiling . . . 79

A.1.3 Compiling and installing the new kernel . . . 81

A.1.4 TPM2 tools Installation . . . 83

A.1.5 Keylime installation . . . 83

A.1.6 Keylime Agent installation . . . 85

A.1.7 Configuring Keylime Agent . . . 85

A.1.8 Installing and configuring Kubernetes . . . 86

A.2 Verifier machine . . . 86

A.2.1 Keylime Verifier and Registrar installation . . . 86

A.2.2 Installing and configuring Kubernetes . . . 86

A.2.3 Proxy Kubernetes Master node . . . 89

A.2.4 Deploy Pod Distribution Service Pod . . . 92

B Developer Manual 93

B.1 IMA patch 93

B.1.1 `ima-slt` template implementation 93

B.2 Filtering Measurement Log 95

B.3 Filtered ML verification 96

B.3.1 New `ImaSlt` Class 97

B.3.2 Modified Entry Class 97

B.3.3 Processing the Measurement List 98

B.4 Pod Distribution Service script 99

B.4.1 Kubernetes Node IP Retrieval 99

B.4.2 Verifier and Pod Mapping Checks 100

B.4.3 Multi-Application Concurrent Execution 101

B.5 Authorised workload deployment 104

Chapter 1

Introduction

The advent of cloud computing signifies a substantial paradigm shift in the management of applications and data. It is widely acknowledged that cloud computing represents a fundamental solution to the problems associated with uncertain traffic peaks, computational overheads and potentially large hardware investments. Indeed, it has the potential to transform the IT industry by rendering software and infrastructure even more attractive as services. This paradigm is a complementary addition to the current internet-based model of consuming and distributing IT services by virtue of the fact that it provides dynamically scalable and often virtualised resources. The fundamental concept underpinning this approach is the delegation of the management and provision of software and hardware resources to external entities specialising in these domains (cloud providers). This strategic decision aims to ensure a superior quality of service at a reduced cost. The increasing demand for processing and storage capacity, precipitated by the Internet of Things (IoT), has further accelerated the adoption of fully virtualised environments. In such environments, workloads from multiple tenants are shared on a common infrastructure [1]. While this model has been demonstrated to increase efficiency and reduce operational costs, it has also been shown to engender new challenges.

To address the unique requirements of the cloud, such as high scalability, reliability and continuous integration and deployment processes, a new paradigm for cloud application development has emerged: microservices design [2]. Microservices-based applications comprise numerous decoupled and autonomous services, each dedicated to the execution of a specific task and developed and implemented independently. These services facilitate the transmission of data or information by providing designated interfaces. In contradistinction to monolithic design, which is characterised by a single code base where all functionality is tightly coupled, microservices architecture enables greater granular scalability and flexibility for developers at minimal cost. It is evident that well-known corporations such as Amazon, Netflix and Uber are transitioning from conventional monolithic designs to microservices in order to accommodate their extensive user bases and meet their stringent scalability requirements. The security requirements of contemporary application container technologies are contributing to the development of more secure and reliable computing environments. This development necessitates a more scalable attestation process that is suitable for microservices architectures or orchestrated container deployments.

The transition to Infrastructure-as-a-Service (IaaS) and microservices architecture confers numerous advantages, including flexibility, availability, cost efficiency, and the capacity to scale applications on demand. However, this paradigm shift also introduces significant new security challenges. The off-premises nature of computing resources invariably results in a diminution of control over data and resources for the organisation. The level of security and protection is not guaranteed. In multi-tenant environments where workloads are shared, ensuring the integrity of those workloads and protecting sensitive data from unauthorised access becomes critical. The cloud computing market is characterised by a paucity of regulation, with service level agreements (SLAs) frequently ambiguous. This has the effect of making it difficult to hold providers accountable for security breaches. Some researchers have noted that the isolation of virtual resources remains an ongoing challenge. It is important to note that resources which are potentially implicitly shared between virtual machines (VMs), such as the last-level cache on multicore processors and memory bandwidth, present a risk of either security or performance degradation. It has been hypothesised that incorporating security constraints and performance isolation as part of SLAs in future cloud computing environments would enhance the visibility of cloud resources. It is important to note that the cloud environment is not without its specific challenges. These include the management of insider threats and the guarantee of transparency about security and residual risks [1].

In order to provide effective support for microservices architecture in the cloud, lightweight virtualisation has become a key element. Containerisation is a form of virtualisation that enables resource isolation with low overhead by sharing the host operating system kernel. Containers have been shown to be comparatively lightweight, simple to manage and offer superior performance in comparison to traditional virtual machines (VMs). These systems have been shown to require less storage and to reduce infrastructure costs. Docker is an example of an emerging containerisation technology that is gaining popularity over VMs [3]. This technology has become the preferred method for supporting microservices architecture, particularly due to its inherent scalability and redundancy mechanisms.

To overcome the inherent security challenges posed by cloud environments, with a particular focus on the verification of the integrity of systems running on shared infrastructures, the concept of trusted computing is of considerable importance. The Trusted Computing Group (TCG) [4] has developed a set of standards known as Trusted Computing, the purpose of which is to establish trust in a platform by identifying its hardware and software components. The purpose of this process is to guarantee that a computer exclusively operates with software that has been verified as trustworthy. The Root of Trust (RoT) is an essential component in the development of trust-based systems, and it is required to function in accordance with expectations, as any deviation in behaviour would go undetected. Depending on the requisite security level, the RoT can be implemented in either software or hardware, and it serves as an intrinsically trusted starting point from which a chain of trust is established. This trust is established during the system's inaugural startup phase, with the first software component to execute being an element that itself cannot be measured or validated. From this initial point, trust is progressively transferred from one component to the next as each assumes control, thereby ensuring the integrity of the platform is maintained throughout its operation. The employment of certificates within this specific context functions to provide assurance with regard to the integrity of the implementation process, through the demonstration of the RoT's construction itself.

The present paper sets out the ways in which mechanisms such as Measured Boot and Remote Attestation serve to extend trust into the operating system and its applications. Remote attestation is a process that enables a third party to verify the integrity of a system. This is achieved by analysing measurements (i.e. file hash values) recorded by an Integrity Measurement Architecture (IMA) log. The Trusted Platform Module (TPM) is a hardware component that, when utilised in conjunction with the IMA, engenders a more robust, hardware-based chain of trust. Trusted Computing, and more specifically the attestation procedure, facilitates the establishment of trust in data and applications in cloud scenarios by means of providing technical mechanisms that enable the verifiable enforcement of control over the hardware and software that access sensitive workloads. Attestation can thus be defined as the mechanism that engenders trust in the domain of confidential computing. Attestation is the mechanism that establishes trust in confidential computing and forms the basis of security guarantees that enable workloads to execute securely when integrated into potentially untrusted cloud environments [5].

Container-based virtualisation remains the preferred method for microservices architecture due to its inherent scalability and redundancy mechanisms. In the context of a containerised cloud environment in which thousands of applications and microservices have been deployed, the necessity for an automated system to manage, scale and orchestrate this ecosystem is evident. It is widely acknowledged that Kubernetes (K8s) [6] is the most prevalent orchestrator for microservices in a containerised architecture. The platform is characterised by its portability, extensibility and open-source development, which facilitates its application in the management of containerised workloads and services. In the context of orchestration, the utilisation of Kubernetes is indispensable for the automation of tasks essential to the management of workload connections and operations.

Although significant progress has been made in the fields of Trusted Computing and orchestration, the adoption of multi-tenant cloud environments, particularly those employing microservices architectures on Kubernetes, gives rise to intricate security and privacy concerns. While remote attestation is imperative for the purpose of verifying integrity, there is a possibility that sensitive information may be unintentionally revealed in Measurement Logs (ML) pertaining to unrelated applications or pods, especially in multi-verifier environments. In such circumstances, the confidentiality of logs is diminished when events within the logs can be identified without direct access, and the enforcement of fine-grained access controls becomes more challenging. Consequently, it is advisable to ensure that tenant-shared measurement logs cannot easily be predicted in order to avoid revealing underlying occurrences.

The central theme of this thesis is the exploration of challenges, with a particular focus on privacy concerns in the context of remote attestation of pods within Kubernetes environments. The objective is for verifiers to receive only the specific attestation data relevant to their task, without exposing the broader contents of Measurement Logs. The proposed thesis work is an enhancement to the traditional attestation framework, incorporating privacy-preserving capabilities. This includes the development of a new IMA template, the implementation of which is intended to ensure unpredictable measurements by adding a nonce field to each entry and thereby enforcing discretion in log access. The solution is predicated on the utilisation of the Kubernetes Master node as a proxy to manage access control, thereby ensuring that verifiers only access data relevant to their attestation tasks, while preserving the secrecy of other system details. The primary objective is to ensure the effective safeguarding of privacy, the mitigation of the risk of unauthorised data exposure, and the assurance of secure pod attestation, whilst ensuring the anonymity of cluster details. The solution has been developed by modifying Linux IMA, the Keylime framework, and adding Kubernetes capabilities.

Chapter 2

Trusted Computing and Trusted Platform Module TPM

Trusted Computing [5] is a hardware and software design approach developed by the Trusted Computing Group (TCG) [4] to improve the security of personal computers and billions of endpoints. The TCG outlines methods for establishing trust in a platform by identifying its hardware and software components. The technology ensures that a computer runs only trusted software and communicates exclusively with other systems operating trusted software.

The term 'trust' is predicated on the expectation of predictable behaviour; however, this does not necessarily imply that such behaviour is in itself trustworthy. To evaluate the expectations of a platform, it is first necessary to define the term in question by reference to its actions. It can be hypothesised that, if the components of the system in question consist of hardware and software, and operate similarly, can be deduced that there should be equivalence in their trust properties.

Trusted Computing is predicated on the integration of hardware and software mechanisms for the purpose of ensuring and validating the system's integrity. A trusted platform is expected to act consistently with its intended design during any operation. Consequently, its actual behaviour must be examined against the predicted outcomes to ascertain its reliability.

A Trusted Platform [7] is built on protected capabilities and shielded storage. Protected capabilities are fundamental operations performed by hardware and firmware that are critical to the trust of the entire TCG subsystem. These capabilities rely on shielded memory locations and special areas designed to securely store and process sensitive data.

2.1 Trusted Computing Group

The Trusted Computing Group (TCG) is a consortium of various technology companies which aims to enhance the security that could be incorporated into future generations of computers. It establishes new specifications and standards for Trusted Computing and supports interoperability between different companies' products.

One of the most significant contributions was the creation of the Trusted Platform Module (TPM) [8] is a hardware-based security chip that meets the TCG's specifications, integrating 'roots of trust' into computer systems. Beyond developing the chip, TCG has also established other hardware-based security techniques, such as secure boot and remote attestation.

2.2 Trusted Computing Base

A Trusted Computing Base (TCB) [5] is the set of components of a computer system responsible for enforcing security policies and protecting sensitive data. The TCB, which defines the system's security region, encompasses hardware, firmware, and software critical to protection. Even if other system components become compromised, the TCB must remain secure to safeguard the system. The TCB must be designed to withstand breaches in other system components without compromising overall security.

2.3 Root of Trust

Trusted Computing begins with a *Root of Trust (RoT)* [9], which is an intrinsic trusted component for ensuring trust in a platform. It must consistently behave as expected since any misbehaviour would go undetected.

Although it is not possible to explicitly verify the correctness of the behaviour of RoTs at runtime, certifications can provide confidence in their trustworthy implementation by knowing how it was built. In other words, certificates serve as proof that the RoT has been built in a way that ensures its trustworthiness [5]. It can be used in testing environments to issue a certificate indicating the TPM's Evaluated Assurance Level (EAL), which assures the proper implementation of its RoTs. The first example of the root of trust is the *Core Root of Trust for Measurements (CRTM)*, a static root of trust (SRTM) located in the BIOS, which represents the first software to execute and cannot be measured or validated. To safeguard the CRTM from attacks, the platform manufacturer can make it immutable by placing it in ROM. Its certificate can confirm that the TPM is correctly integrated into a motherboard.

Rot can be either software or hardware [9], depending on the required level of security. Still, hardware is typically preferred in modern systems as it is more resistant to attacks. To achieve high trust, the system's hardware must be designed to ensure that the integrity of the hardware, software, and data remains uncompromised. This requires the hardware to support continuous monitoring to maintain the integrity of both hardware and software during power-up, initialisation, and runtime operations.

In case of multiple RoTs involved, it is opportune to establish a transitive *chain of trust*. It builds on the trustworthiness of the RoTs as the starting point for a chain of trust, where trust is passed progressively from one executable function to the next as each takes control of the system.

The TCG requires three different kinds of RoT in a trusted platform:

- **Root of Trust for Storage (RTS)**: The RTS is the shielded memory location, the key concept of a trusted platform used to protect keys and data. It securely stores integrity measurements and keys used for cryptographic operations. Integrity measurements are generally considered non-sensitive information and can be safely shared using RTR specifications. On the other hand, the keys used are asymmetric keys with a private part, so TPM rejects access and manipulation without proper authorisation.
- **Root of Trust for Measurement (RTM)**: The RTM oversees the process of verifying system integrity and sends the resulting measurements to the RTS. Those results are the evaluations of the system's state taken from the initial boot stage, starting from the CRTM. The CPU typically begins by executing the CRTM to initiate the measured boot process, in figure 2.1 and records all subsequent system activity.
- **Root of Trust for Reporting (RTR)**: The Root of Trust for Reporting (RTR) is a reliable source for accessing information stored in the RTS. RTR securely accesses data held from the RTS and transmits them to external entities to verify their authenticity. Before transmission, the RTR generates an accurate digest of the data stored in shielded locations and digitally signs it to prove identity and integrity. These signed digests are exported as *integrity reports (IRs)*.

Each root is inherently reliable and doesn't require external monitoring.

At least one hardware component should act as the Root of Trust in the system. The hardware Root of Trust and its associated firmware or software are responsible for maintaining system trust and protecting against tampering. The Trusted Platform Module (TPM) is a commercial implementation of this technology, incorporating RTS and RTR features. However, the TPM itself doesn't guarantee the implementation of an RTM, as it's designed to be a passive module [See PCR section].

2.4 Trusted Platform Building Blocks

Trusted Building Blocks (TBB) [10] are components of the Roots of Trust that lack shielded locations capabilities. They typically include instructions for initialising the RTM and TPM and are platform-specific. An example of a TBB is the combination of the CRTM, its storage connection, the TPM's motherboard connection, and physical presence detection mechanisms. The TBB has a one-to-one logical relationship with the PC Motherboard. It plays a crucial role in the initial phase of the measured boot process (Figure 2.1), which includes, together with a RoT, the CRTM and serves as the foundation of a chain of trust. TBB should be designed to prevent devices with additional measurement codes from unintentionally expanding the TBB boundary without verifying the connections' trustworthiness.

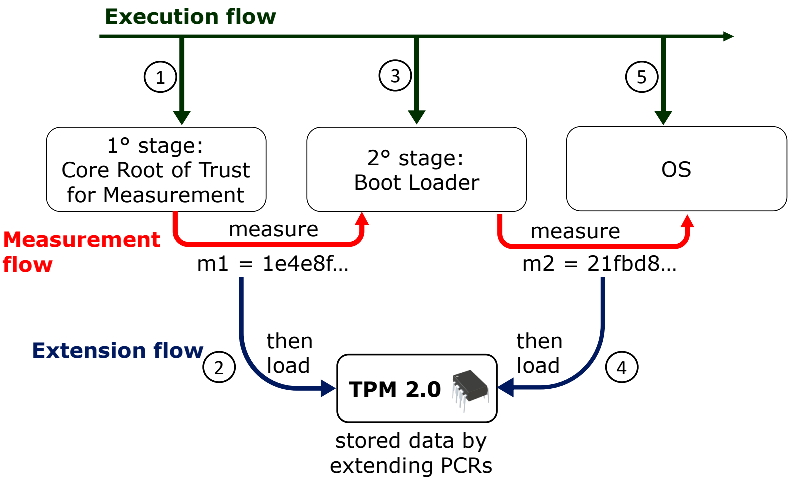


Figure 2.1. Measured boot from a static root of trust [11].

2.5 Boot Types

- **Plain boot**: there's no specific security requirement
- **Secure boot**: It is a hardware-based security feature where the firmware ensures that only trusted digitally signed software is loaded during system startup. If the initial verification fails, the platform is stopped to prevent unauthorised or untrusted software from running.
- **Trusted boot**: It is a software-based security feature where the Operating system (OS) verifies the digitally signed components (i.e. drivers, antimalware). If this verification fails, the platform is stopped during the OS operational state.
- **Measured boot**: in figure 2.1, the system records the execution of each component during the boot process. It calculates a digest for every piece of code, starting from the firmware, without interrupting the boot. This process leverages the transitive trust established from the static CRTM to build a chain of trust.

2.6 TPM 2.0

The computer engineers [8] who founded the Trusted Computing Group (TCG) aimed to address the decline in PC system security by creating a hardware-based foundation for building secure systems. A Trusted Platform Module (TPM) is a specialised hardware component intended to be physically integrated with a computer's motherboard to establish secure operations instructed by Trusted Computing. A TPM safeguards user identity and data by securely storing encryption, decryption, and authentication keys, providing robust protection against unauthorised tampering.

Version 2.0 is the latest update of the module, replacing the previous version 1.2 to enhance security and integration with portable pc. It has replaced version 1.2 and is the standard version implemented in all modern devices. This thesis focuses on version 2.0, so all described functionalities pertain to it.

The TPM is a separate component from TCB [5], which can determine if the TCB's integrity has been compromised. In some uses, the TPM can help prevent the system from starting if the TCB cannot be adequately instantiated. It works as a "passive" entity, a dependent component that relies on the CPU for activation and operation.

2.6.1 Architecture

The primary components can be observed in the architecture depicted in figure 2.2.

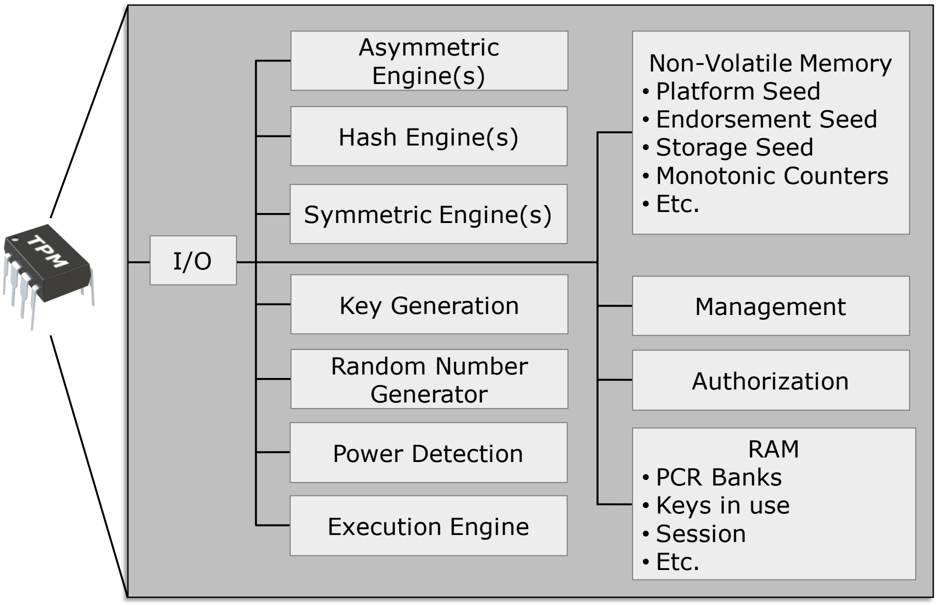


Figure 2.2. TPM 2.0 Architecture (Source [5]).

2.6.2 Main features

- **Low-cost chip**: a cost-effective component that can be easily integrated into most servers, laptops, and PCs.
- **Tamper-resistant**: although not wholly invulnerable to physical tampering (tamper-proof), it offers significant resistance to specific hardware manipulations.

- **Hardware random number generator**: a true random-bit generator [10] used to seed random number generation. The Hardware random number generator (HRNG) is a non-deterministic source of randomness in the TPM. It is used for key generation, nonce creation, and strengthening passphrase entropy.

- **Generation of cryptographic keys**: cryptographic keys can be generated using the HRNG for encryption purposes. The resulting secret key values are stored in a shielded location and are intended for limited use.

- **Remote attestation**: support the attestation procedure reporting platform state, explain better in chapter 2.7. It operates as an RTS and RTR component, collecting and securely reporting a hash summary of the software configuration to an external third party to prevent unauthorised modifications and ensure that the system is operating as expected.

- **Binding**: data encrypted with a TPM secret key is binding to the specific TPM module. It uses an internally secret TPM bind key, a unique RSA key derived from the storage key. This prevents unauthorised decryption outside the TPM itself, but it can introduce complexities when data needs to be exported to other devices.

- **Sealing**: is a security mechanism that ties decryption to the data and the specific TPM state at the time of encryption. It is mandatory to have a platform state with authorised access to keys and data. This includes the state of all running applications and configuration files, providing an extra layer of protection.

- **Hardware device authentication**: relies on the Endorsement Key (EK), a unique secret key burned into each TPM chip at creation. This EK can be used to identify the specific device.

2.6.3 Securely storing data

The TPM provides secure data storage through physical and cryptographic isolation. In the first case, data is stored in the TPM's limited non-volatile RAM with strict access controls. On the other hand, keys are stored outside the TPM and encrypted with a TPM-held key, preventing data migration. Both physical and cryptographic enforce mandatory access controls to protect data.

2.6.4 Implementations

- **Discrete TPM**: Discrete chips [12] are designed, manufactured, and rigorously tested to ensure they are highly resistant to tampering. This includes protection against sophisticated attacks like probing and freezing.

- **Integrated TPM**: a step down in security compared to standalone TPMs. While they still have a hardware TPM component, they're integrated into other chips, making them less tamper-resistant. Indeed, this level is not designed to be tamper-resistant.

- **Firmware TPM**: is a software implementation of the TPM that runs on the main CPU in a protected environment called a Trusted Execution Environment (TEE).

- **Hypervisor TPM**: runs in an isolated execution environment and acts as virtual TPM. Security is comparable with firmware TPM.

- **Software TPM**: software emulations of hardware TPMs running in user space. They are helpful for testing purposes but more vulnerable to attacks.

2.6.5 TPM Platform Configuration Register PCR

PCRs (Platform Configuration Registers) [5] [8] are shielded locations used to validate the integrity of a system's boot process, from BIOS to the application, passing for security-critical configuration files. They record events that can affect the system's security state and store these records in a log. More precisely, they store measurements digested by external software. If the registers provide a trustworthy representation of the software's behaviour, then all the registered values can be trusted.

While it's technically possible for a single PCR to record all log entries, using multiple PCRs simplifies evaluating different stages of platform evolution during the boot process. After the recorder process, the TPM report these measurements by digitally signing them with a specific key. In other words, TPM may provide an attestation of the value in the PCR, which, in turn, verifies the contents of the log.

Standards mandate that TPMs have at least 24 Platform Configuration Registers (PCRs), each storing 20 bytes of data. The first 8 PCRs are reserved for TPM-specific use, while the remaining ones are available for the operating system and applications. The table specifies the PCRs allocation (table 2.1).

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

Table 2.1. Standard PCRs allocation in a system.

PCRs may be implemented in volatile or non-volatile storage. However, their temporal values must be reset after each system reboot or specific signal. If the registers are not reset, previous integrity measurements might remain in the PCRs, potentially corrupting the validation process. Resetting the registers involves setting their values back to 0.

Callers cannot directly change PCR values. The TPM manages the modification of PCR values through the most used operation called "extend" (equation 2.1), which takes the old PCR value and concatenates some data to be extended. So, data sent to the TPM are collected via cumulative hash inside PCRs.

$$\mathrm{PCR}_{\mathrm{new}} = hash(\mathrm{PCR}_{\mathrm{old}} \,||\, digest\_of\_data\_to\_extend) \tag{2.1}$$

The TPM can only modify PCR values during the "extend" operation or system reset. Once a PCR value is reached, it's nearly impossible to determine the exact sequence of measurements contributing to it. This makes it challenging to recreate the same PCR value using different measurements.

2.6.6 Secure Identity

As discussed so far, the TPM provides information about the platform's integrity state by quoting PCR values. Potential external entities that verify the platform state need to identify the TPM that issued the quote and ensure a binding between the RTR and the RTM that performed the measurements. This identification is achieved using asymmetric keys called Endorsement Keys (EKs) derived from a unique endorsement seed within the TPM. All EKs generated from the same seed are tied to the same TPM and RTR. These keys are non-migratable, ensuring they remain exclusively within their originating TPM. Since the seed is statistically unique to each TPM, there is a very low probability of having two TPMs with the same EK.

Trust in an EK is typically established through a certificate that confirms its authenticity. However, because EKs uniquely identify a TPM, directly using them for operations like signing or encrypting could lead to privacy issues, such as enabling activity tracking and data aggregation that could compromise user privacy. To overcome this, the TCG recommends limiting EK usage and promoting using other dedicated keys, such as *Attestation Identity Keys (AIKs)*. This key is generated from the EK, which ensures device-specific anonymity. AIK is used to sign specific data, such as the digest of PCR content, during quote operations and for authentication at the verification phase. An Attestation CA must be used to certify that a given AIK has been generated by a valid TPM. This CA maintains a direct binding between AKs and EKs. It also permits users to have the option to select a CA that guarantees it will not track these associations, ensuring greater privacy [5].

2.7 Remote Attestation (RA)

Remote attestation is a process through which a system, known as the *attester*, generates cryptographically signed evidence detailing the software it has run and its accessed files. This evidence is then sent to another system, called the *Verifier*, which checks the validity of the information and evaluates the status of an untrusted remote device. The outcome of this verification is a cryptographically signed statement confirming the integrity of the attester system.

The use of a reliable Measured Boot process, which is typically implemented through the presence of a Root of Trust for Measurement (RTM), a Trusted Platform Module (TPM), and TPM-compatible boot components [13], is one scenario for remote attestation, but it is not the only possible approach. In this context, the TPM is instrumental in ensuring the integrity and trustworthiness of the target device's platform, thereby establishing a foundation for trust between remote devices.

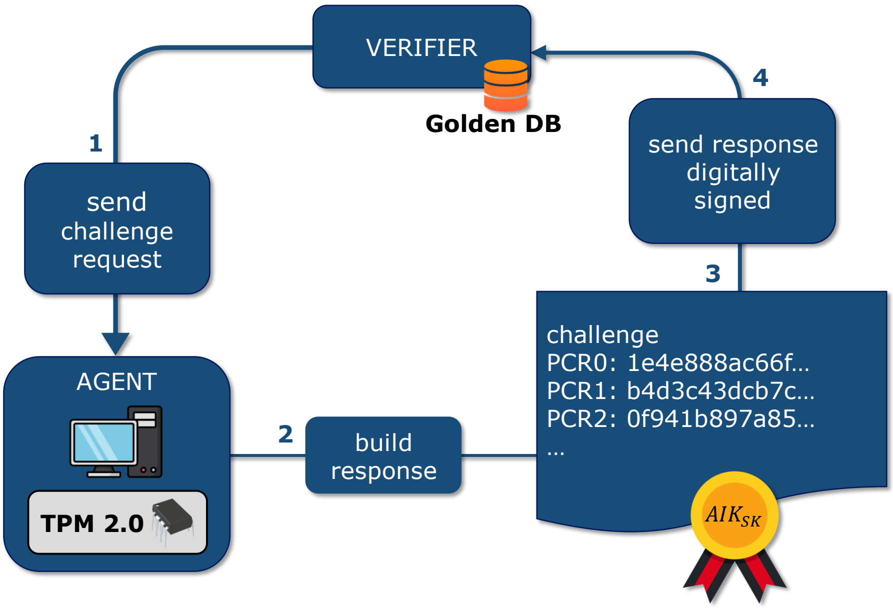


Figure 2.3. Remote attestation Schema.

2.7.1 Execution flow

The RA's process can be described, following the 2.3, through these steps:

1. Verifier sends a request called "Challenge Request" to the attester machine specifying which PCR values must be attested. Usually, to avoid replay attacks and ensure freshness, it also incorporates a *nonce* (unrepeatable random number) in the request's body.

2. When the attester receives a challenge request, it leverages the RTR and RTS capabilities provided by TPM to retrieve the PCR values stored in shielded locations as an integrity report. It then builds a response (or quote), including the nonce from the request and all requested PCR values. However, to ensure hardware device authentication, the response must be digitally signed with the private part of the AIK.

3. The attester sends the digitally signed response to the Verifier. If a measurement log is requested, the Verifier must check its integrity. This process is more complex and is described in detail in Chapter 3.5 in Linux implementations.

4. Before proceeding, the Verifier ensures the response is authentic and up-to-date by validating the signature. To do this, the verifier consults its list of public AIKs for the involved devices. It then assesses the attester's trustworthiness by comparing the verified quote to a known trusted reference stored in a golden DB, known as *golden values*. A match indicates that the device is trustworthy and is operating as expected. Ultimately, the remote device issues an attestation result, indicating whether the target device is trusted based on the trust assessment.

Chapter 3

Integrity Measurement Architecture (IMA)

As explained in the previous chapter, TCG specifications outline the foundational components of a Trusted Boot. Specifically, the CRTM initiates the RTM, which measures the system's boot components and securely stores these measurements in the RTS (in the TPM's PCRs). This allows an external party to verify that the system has been booted securely. Ideally, the chain of trust would extend to the application layer. Still, operating systems handle various types of executable content (e.g., kernel, modules, binaries) in an unpredictable order, and TCG's specifications are OS-agnostic and do not specify how the RTM extends into the operating system.

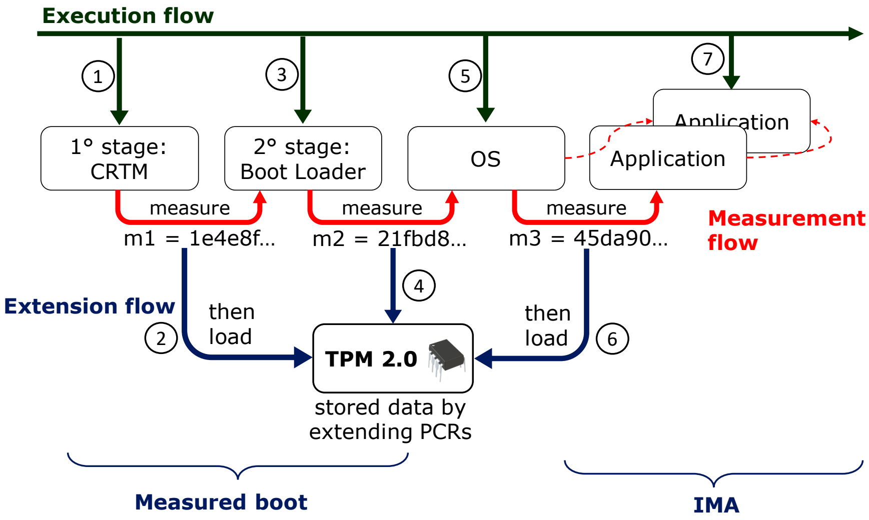


Figure 3.1. Measured boot from a static root of trust to the application layer. [11].

To be precise, figure 2.1 doesn't explicitly show the whole process. Figure 3.1 provides a more comprehensive overview of the Measured Boot process from the BIOS to the application layer.

3.1 Overview

The Integrity Measurement Architecture (IMA) [13] is the Linux kernel's implementation of the TCG's integrity measurement system (so as RTM) integrated into the Linux kernel since 2009 and extends the chain of trust from the BIOS to the application layer. IMA is a leading TCG-compliant solution for measuring dynamic executable content. It extends the principles of Measured Boot and Secure Boot to the Linux OS and protects system integrity by collecting, storing, and locally evaluating measurements. The platform needs a Measurement Log (ML) file to record the sequence in which events occur.

3.2 Design

The design of the IMA module has been implemented with three components which cooperate to enable the RA process [14]:

- The **Measurement Mechanism** on the attested system identifies which component of the run-time environment needs to be measured, decides the measurement time, and ensures protection on ML and secure storage. This mechanism starts with a base measurement when a new executable is loaded and extends to other executable content and sensitive data files. Initially, the BIOS and bootloader measure the kernel code, then the kernel measures changes to itself and user-level processes. It stores the measurement log as a list in the kernel, and the first entry is always the *boot_aggregate*, confirming the SRTM's correctness. The operations used in this protocol are customized by policies explained in section 3.3.
- The **Integrity Challenge Mechanism** enables a remote verifier to request the measurement list along with a TPM-signed summary and check them later with specific trusted values. Since this protocol involves remote entities, it is not free from threats. Therefore, communication should occur over TLS, and the system must adhere to remote attestation protocols, using a nonce and digital signature for security. This permits freshness, authenticity, and integrity inside this mechanism and ensures the system's trustworthiness under examination.
- An **Integrity Validation Mechanism** is the last mile mechanism allowing the verifier to validate the received information and assess the trustworthiness of the attesting system's run-time integrity. Each measurement list entry is tested against a list of trusted values to determine whether to trust or distrust the attesting system. The idea is that the entry matches some predefined value with known integrity semantics, which the challenging party has already measured and incorporated into the golden database.

To be precise, to extend Secure Boot and Measured Boot to the operating system and its applications, IMA provides two mechanisms called Appraisal and Measurement [13]. The first enhances Secure Boot, whereas IMA-Measurement extends Measured Boot into the operating system and is more aligned with this thesis's main concept.

3.2.1 IMA-Measurement

IMA measurement extends the idea of Measured Boot by integrating it into the operating system. This integration enables Remote Attestation, allowing a third party to challenge the system and verify its integrity. To achieve this, IMA keeps a sequential list of hash values within the kernel, which includes all files measured since the OS took control. Although IMA can operate without a Trusted Platform Module (TPM), it is often used in conjunction with one to create a stronger, hardware-based chain of trust. As a result, the measurement list plays a crucial role in remote attestation, as it provides a detailed record of all software executions for an external party that will be asses. This remote party can then compare this list to a database of established valid reference measurements to identify deviations from the expected integrity state.

3.3 Policies

IMA Policies specify the criteria and rules for determining which files should be measured, ensuring that only relevant and essential files are included in the integrity measurement process. IMA includes three built-in policies that are configured using kernel boot parameters:

1. **Tcb policy**: enables the measurement of different components, such as kernel modules, executed software, files loaded into memory using mmap, and files that the root user opens for reading;
2. **Appraise_tcb policy**: evaluates the same components as the Tcb policy in real-time, rather than just measuring them. It denies access if the files do not correspond to their recognised good hash values.
3. **Secure_boot policy**: This policy solely evaluates the kernel, its related modules, and the IMA policies.

3.4 IMA Templates

The Measurement Mechanism fills [15] the measurement list using an ad-hoc template. However, whenever a new template is introduced, the functions responsible for displaying the measurement list must adapt to the new format, potentially leading to excessive code growth. This issue is addressed by decoupling template management from the rest of the IMA code.

Each template defines the fields that describe the event triggering the measurement. In particular, these events are important in the data structures used in the managed mechanism. The structure consists of the *template descriptor*, which specifies the information to be included in the measurement list, and the *template field* responsible for generating and displaying data of a specific type.

The original IMA template has a fixed length and includes the *filedata-hash* and *path-name*. The file data hash is restricted to 20 bytes calculated with common hash algorithm such as Secure Hash Algorithm 1 (SHA-1), while the path name is a null-terminated string limited to 255 characters. Enhancing the current version of IMA by creating additional templates is essential to exceeding these limitations and incorporating extra file metadata.

PCR	template-hash	template-name	filedata-hash	filename-hint
10	b926a[...]36ed8	ima-ng	sha1:999[...]e5f	boot_aggregate
10	dc3b9[...]5f88b	ima-ng	sha1:918[...]a10	/usr/bin/kmod
10	dc3b9[...]5f88b	ima-ng	sha1:1ba[...]0ef	/usr/bin/cp

Table 3.1. Example of IMA Measurement Log created with ima-ng template

Table 3.1 illustrates an example of a Measurement Log (ML) generated using the default template ima-ng. In order, the fields represent:

1. the index of the PCR where the entry was extended, which in this instance is PCR 10;
2. the *template-hash*, representing the digest extended in the specified PCR calculated over the template fields using the SHA-1 algorithm;
3. the *template-name* associated with the entry;
4. the filedata-hash, which is the hash derived from the contents of the file or the boot PCRs in the case of boot_aggregate in this example, SHA-1 is the algorithm employed, as it is the default hashing method;
5. the *filename-hint*, typically indicating the file's path.

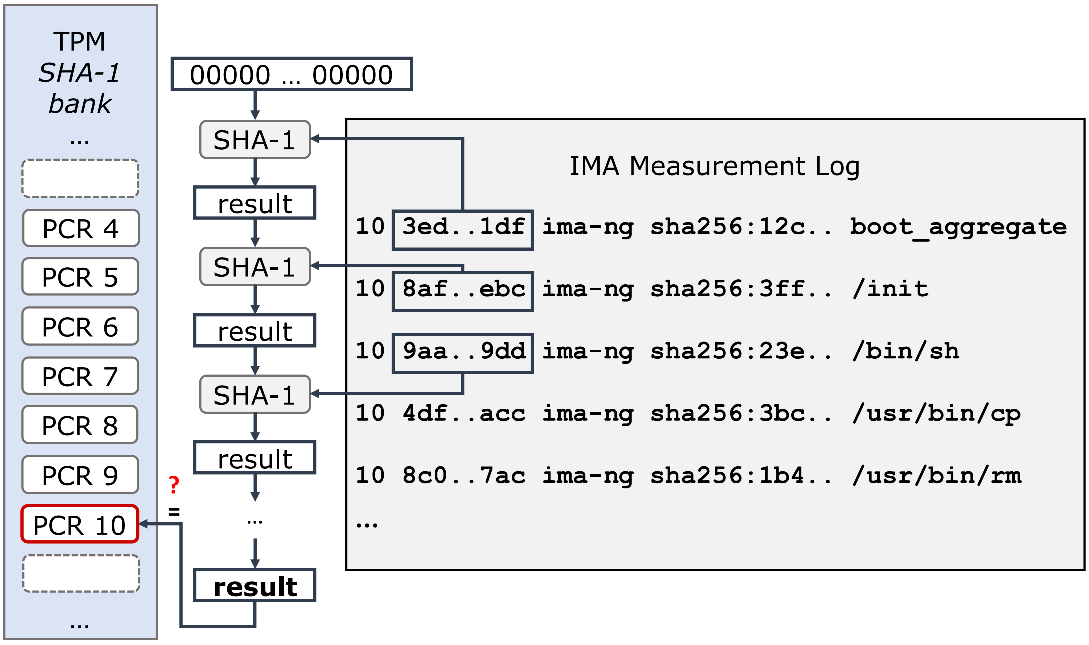
3.5 ML Integrity Verification and Entries Validation

The verifier initiates the IMA Integrity Challenge Mechanism by challenging the attester, which starts the Remote Attestation process. This process generates an integrity report containing the signed TPM quote and the measurement list. The verifier first verifies the signature and then activates the Integrity Validation Mechanism, performing two main processes.

The first is the ML verification process (figure 3.2), which examines the integrity reports to determine if they have been tampered with or are outdated, thus providing evidence that the system lacks trustworthiness. This process begins by retrieving the template-hash field from the first ML entry, corresponding to the boot_aggregate (*th₁*). This template hash is combined with a digest of all zeros, and a new digest is calculated from the resulting concatenation. The extend operation is performed for each entry th_i, where i is a sequential number that increments until the entire ML is processed.

The final result can then be compared with the IMA aggregate stored in PCR 10. The formula 3.1 summarizes the entire operation:

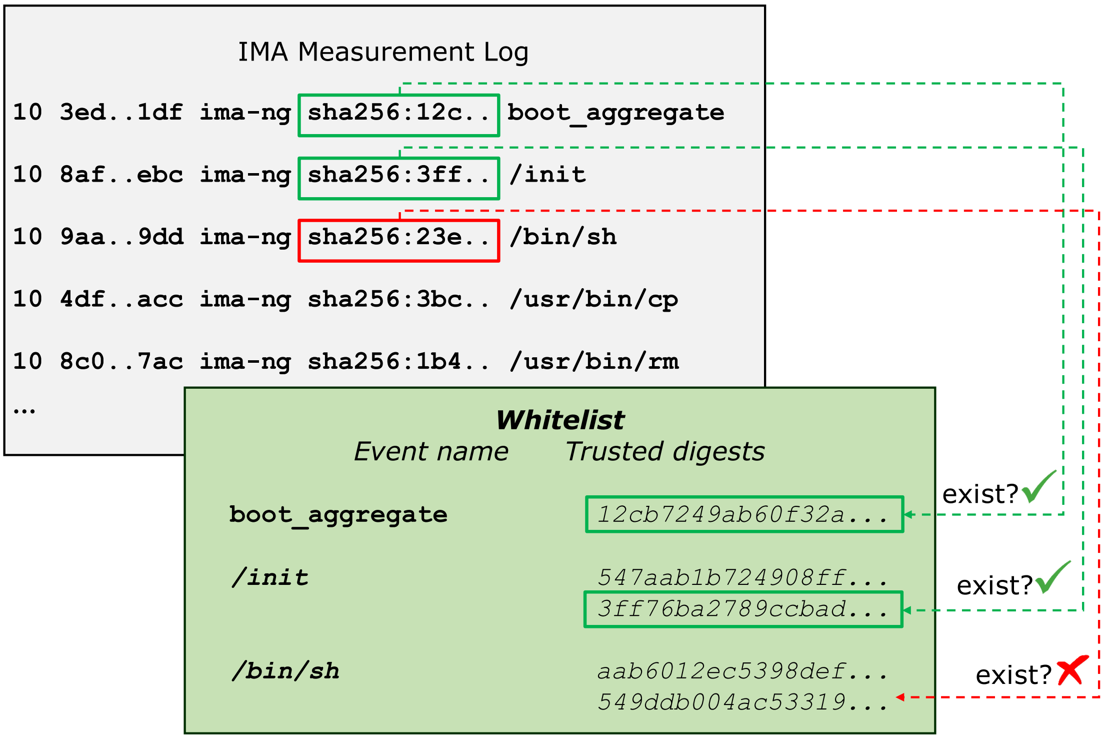
$$\mathrm{PCR}_{10} \;=\; SHA1(\ldots SHA1(\,SHA1(0\,||\,th_1)\,||\,th_2)\ldots||\,th_i) \tag{3.1}$$



```
10 3ed..1df ima-ng sha256:12c.. boot_aggregate
10 8af..ebc ima-ng sha256:3ff.. /init
10 9aa..9dd ima-ng sha256:23e.. /bin/sh
10 4df..acc ima-ng sha256:3bc.. /usr/bin/cp
10 8c0..7ac ima-ng sha256:1b4.. /usr/bin/rm
...
```

Figure 3.2. ML verification process (source: [1])

The second process involves assessing the ML entries (figure 3.3). In this context, an entry refers to the file-hash field within the ML that corresponds to a specific event. Each entry is compared then against trusted values contained in a whitelist. This whitelist is organised with a *file-path* and a set of acceptable (trusted) *file-hash* associated with that element. The system is deemed untrusted if the file path associated with the event is not found in the whitelist or if it is present but does not match any trusted values. If the whitelist contains both the file path and file hash of each ML entry, the system is successfully assessed as trusted.



IMA Measurement Log

```
10 3ed..1df ima-ng sha256:12c.. boot_aggregate
10 8af..ebc ima-ng sha256:3ff.. /init
10 9aa..9dd ima-ng sha256:23e.. /bin/sh
10 4df..acc ima-ng sha256:3bc.. /usr/bin/cp
10 8c0..7ac ima-ng sha256:1b4.. /usr/bin/rm
...
```

Whitelist

Event name	*Trusted digests*
boot_aggregate	*12cb7249ab60f32a...*
/init	*547aab1b724908ff...* *3ff76ba2789ccbad...*
/bin/sh	*aab6012ec5398def...* *549ddb004ac53319...*

Figure 3.3. ML measurements validation

Chapter 4

Kubernetes (K8s)

Cloud Computing [2] has improved applications' development and deployment due to critical demands for greater scalability, maintainability and fault tolerance. Meeting the scaling requirement is problematic for applications that previously relied on a monolithic architecture and cannot leverage all the benefits of a cloud-native approach. A monolithic approach is often chosen to develop simple applications or multiple services bundled within the same single code base, making it challenging for several development teams to coordinate. Today, companies are leaning towards a more recent architecture: the microservice approach, where applications are decoupled into independent services. Each has a micro-functionality with a specific task, and they cooperate to provide the complete service. Since a service contains its own system logic and data storage, it can interact through standardised interfaces or event messages, which allows for deployment via virtualisation.

Microservices [16] can be efficiently packaged into containers and deployed on physical hardware, guaranteeing a suitable software execution environment from developers to end users. In general, containerisation is a virtualisation that achieves resource isolation with low overhead by sharing the kernel with the host OS. The container-based virtualisation approach has been the preferred method to help microservices architecture regarding inherent scalability and redundancy mechanisms for machine failures. It permits the on-demand addition or removal of container instances and a good degree of isolation to avoid influence between services.

In a Cloud Computing environment with thousands of applications, an automated system is essential to determine on which physical machine within the cluster a single container (or microservice) should operate, possibly with one or more instances, and also to oversee and control the entire containerised ecosystem. This system already exists, and it is known as Kubernetes. It is the most used orchestrator for microservices in a containerised architecture.

Orchestration refers to automating the tasks needed to manage workload connections and operations and integrating automated tasks into a workflow to perform specific functions. This is required for these containerised, micro-services-based applications, which communicate via ad hoc functions to help with efficient deployment and management across cloud environments.

4.1 Docker container

Docker [17] is an open-source project that automates the deployment of software applications by providing an additional layer of abstraction and automation of OS-level virtualisation on Linux. It enables developers to build, deploy, run and update an application in a *sandbox* having a package with all of its dependencies into a standardised unit called *container*.

Docker is also a *Container management service* that handles the creation and management of containers. Any other components running on the host OS are categorised as host applications. When a client requests a container to be executed, the container management services locate and load the specified container image, applying the necessary configurations, such as network setup and isolation environment settings.

Unlike virtual machines, containers are lightweight and efficient. They share the same operating system and hardware resources, enabling more efficient use of the underlying system. Docker architecture, as shown in 4.1, consists of one or more containers which share the same below infrastructure and Host Operating system. The Docker engine represents a high-level component that initiates container creation, while Container Runtime manages the container's life cycle on a single physical machine. An example of container runtime is `containerd`.

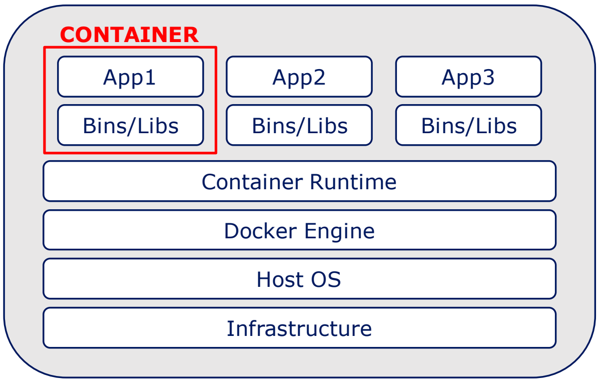


Figure 4.1. Docker architecture

Containers are created starting from images available on Docker Hub, a public repository for Docker images. An image contains executable application code with all the necessary tools, libraries, and dependencies to run the application in a container. When a developer launches the Docker image, it creates one or more instances of that container.

Docker leverages an *automated build* process using a *Dockerfile* to create a single image and launch one container with specified features. A Dockerfile works like a recipe, listing commands in sequence, and developers can use these commands in any combination they need.

4.2 Overview

Kubernetes [6] is a portable, extensible, open-source platform developed by Google in 2014 for managing containerised workloads and services. It is a software orchestrator that facilitates declarative configuration and automation, scaling the entire workload. Its specification is entirely declarative, meaning it allows the description of the logic and application goals without defining how the control flow works.

4.3 State-oriented approach

To ensure the healthy system's functionality, it leverages a *state-oriented approach* through which the system continuously checks if the actual state aligns with customer specifications. On the contrary, try to converge the object state to the expected one. For *object* is intended a *workload resource* explain more in detail in section 4.5. This involves a control loop which also takes care of the desired state at runtime, as shown in figure 4.2.

The Kubernetes controller manager initiates the control loop and enforces the default client configuration. The current state may not correspond with the expected state due to the client updating specifications or internal errors. In both cases, the system reacts accordingly.

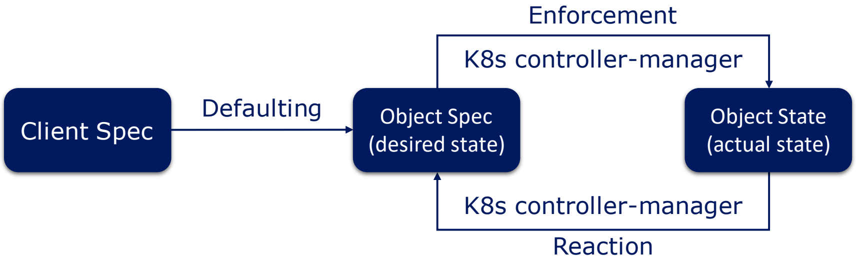


Figure 4.2. Kubernetes control loop (source: [18])

4.4 Architecture

In the Kubernetes architecture, there are different organisation levels. The first distinction to make is for the components involved inside the cluster.

Following the figure 4.3, a brief description can be done starting from the *pod*, which is the smallest deployable computing unit in Kubernetes and serves as Kubernetes' core unit of management for collective task execution. A pod is a wrapper containing one or more connected containers and represents logical boundaries for containers sharing the same context. Each container which belongs to a pod shares the same storage and network resources.

The *Master node* is the authoritative entity responsible for managing the state of the Kubernetes cluster and maintaining its desired configuration. It operates as the Control Plane, which orchestrates the overall functioning of the cluster. The Control Plane performs critical tasks such as monitoring the health of nodes and workloads, scheduling pods to appropriate nodes based on resource availability, and ensuring that the cluster's desired state is achieved and maintained. The Master Node communicates with all other nodes in the Kubernetes cluster to enforce decisions and distribute workloads. It is responsible for Continuous monitoring listen to all changes (i.e. pod deletions, updates, or rescheduling) that can affect the allocation and utilisation of resources, necessitating real-time adjustments.

The *Worker Nodes* execute the containerised applications and workloads. It runs one or more pods scheduled by the master reading configuration written in *YAML configuration files*. These

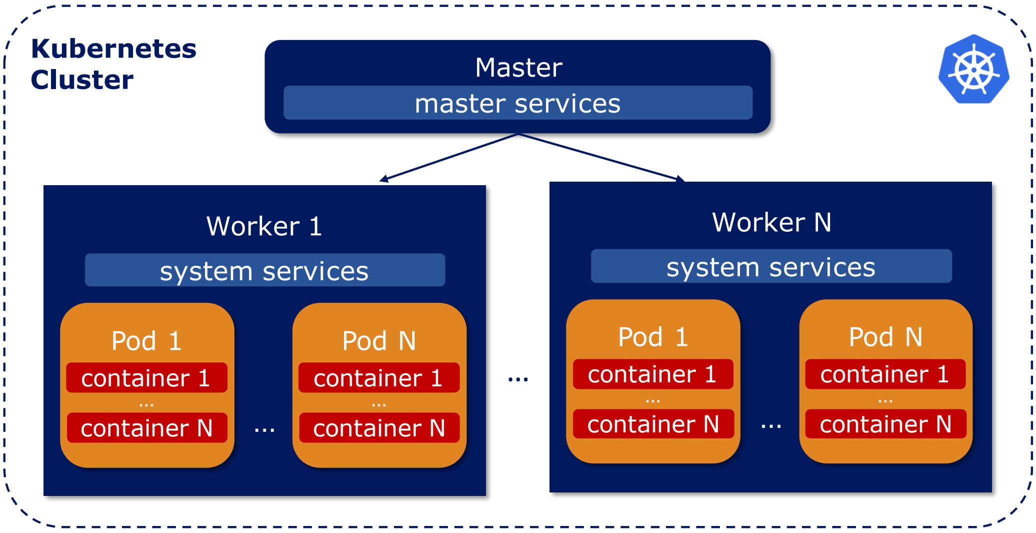


Figure 4.3. Kubernetes cluster

files are heavily used for the workload resources definition and will be explained better in section 4.5

In the end, *Kubernetes Cluster* refers to the set of all described components: Master node, pods and Worker nodes.

As seen in the cluster figure, each cluster has a hierarchy, and each worker node is linked only to its control plane. Figure 4.4 provides a detailed view showing the different node services and how they work and communicate together.

Each master consists of these services:

- **Kube API server**: is the front-end component of the Control Plane through which all interactions pass across it. It exposes all Kubernetes API functions and monitors the cluster status.
- **Kube Scheduler**: the service responsible for scheduling application pods by assigning them to suitable worker nodes based on available resources (i.e. memory, CPU).
- **Etcd**: serves as a database for consistent and highly available key values for all cluster data in Kubernetes. It is usually used to provide data consistency through a backup plan. It stores all resources desired states so the system knows the expected condition toward which the state must move.
- **Kube Controller Manager**: is the control loop component that triggers all the object state modifications of the cluster through controllers. A controller interacts with the Kube API server whenever the current state deviates from the expected one.
- **Cloud Controller Manager**: This service is not used in on-premises or local environments, enabling clusters to connect with cloud provider APIs. It integrates cloud-specific control logic and manages only the controllers specific to a cloud environment, decoupling cloud-related functions from cluster-specific components.

Instead, each worker node consists of these services:

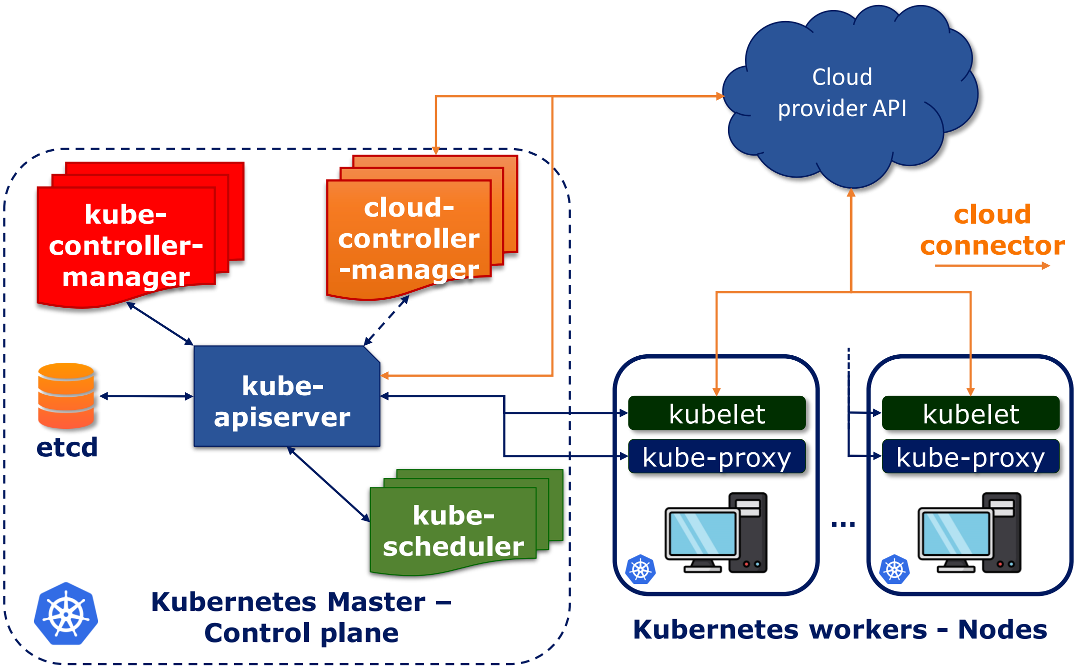


Figure 4.4. Kubernetes Architecture (source: [6])

- **Kubelet**: it's an agent running on the node that monitors both the node's execution state and the status of the pods assigned to it. Kubelet ensures that the containers are run by interacting with the container runtime and keeps the Control Plane updated.
- **Kube proxy**: service which aims to enforce and handle the network rules assigned from that node to pods assigned to it. These rules allow network communication to pods from network sessions inside or outside the cluster.
- **Container Runtime**: an essential component that helps the node run containers efficiently and handle their lifecycle inside the cluster.

4.5 Workload Resources

A resource is a deployable unit managed by controllers. Kubernetes makes available a collection of resources through which a user can declaratively handle the pod's execution. The term “workload” refers to resources with a defined desired state. Every time a resource is modified (since it is stored on the etcd), a controller triggers the event and sees to adjust the current state.

All *Workload Resources* use YAML configuration files and follow the same syntax with standard fields such as: “apiVersion” and “kind” that univocally identify the object, “spec” and “status” which describe the expected and the current resource state respectively.

4.5.1 Deployment

Deployment is a high-level resource provided natively by Kubernetes that describes an application's desired state. In particular, it permits writing the output lifecycle applications in a YAML file, specifying the features of each pod without needing to define how it operates.

Initially, the expected number of replicas (pods) is specified, and the system ensures horizontal scalability to balance the workload. Next, container details such as name, image, and ports are defined, allowing the system to run multiple containers with these configurations. Ultimately, the controller watches the updated file in the etcd, starting the control loop and converting this file into API requests by `kubectl`, the Kubernetes command-line tool.

4.6 Custom Resource

Thanks to its extensible nature, Kubernetes allows adding ”custom resource definitions” (CRDs) to introduce extra functionality beyond its core features. *Custom resources* are extensions of the native Kubernetes API with custom features not included in a standard installation. CRDs permit customisation of cluster functionality and definition of new workload resources following modular structure. For instance, many core Kubernetes functions are now built using custom resources, adapting specific requirements to containerisation applications and enhancing Kubernetes' modularity.

Chapter 5

Keylime Framework

Keylime [19] is a “Cloud Native Computing Foundation (CNCF)” hosted open-source project initially developed by the security research team at MIT’s “Lincoln Laboratory”. It provides an end-to-end scalable solution for remote boot attestation and runtime integrity measurement and represents an easy method of integrating trusted computing into cloud infrastructure. Keylime also permits users to monitor remote nodes using a hardware-based cryptographic root of trust with a periodic attestation procedure.

5.1 Overview

Before discussing Keylime’s specifications, it’s helpful to consider its use context. The framework is placed between trusted and cloud computing, looking to satisfy the ”IaaS” requirement for establishing data and application trust in a cloud scenario.

Infrastructure as a Service (Iaas) is a cloud service model in which essential computing resources (i.e. network, computing, storage) are made available for customers to deploy and run various software, including operating systems and applications. In this model, IaaS resources known as cloud nodes can be physical machines, virtual machines, or containers. Cloud users (tenants) do not manage the underlying infrastructure but have control over the operating systems, storage, and applications they deploy.

However, a cloud service provider lacks the essential building blocks to establish the system’s trustworthiness and protect sensitive data. Furthermore, a tenant cannot verify the underlying platform during cloud deployment and check whether it remains trusted during computation. The TPM module could provide a solution for bootstrapping trust and detecting changes, but unfortunately, it has not been widely adopted in IaaS cloud environments. This is principally because most IaaS services rely upon virtualization, which separates cloud nodes from the underlying hardware on which they run. It is also a cryptographic co-processor, introducing substantially inefficient performance beyond its complexity.

Here, Keylime comes into play. It provides an end-to-end solution for bootstrapping hardware-rooted cryptographic identities into cloud nodes. It permits system integrity monitoring of those nodes via periodic attestation, combining integrity system deviation with identity cloud node revocation. It supports these functions in both bare-metal and virtualized IaaS environments, reducing the cloud provider’s trust responsibility. Additionally, Keylime service has been integrated with software standards to IaaS cloud deployments in non-trusted-computing aware devices. It supports scalability by monitoring thousands of IaaS resources simultaneously as they are elastically instantiated and terminated.

Keylime’s central concept is to incorporate Trusted Computing into high-level security services by using integrity measurements to establish and revoke identities on cloud nodes. This permits higher-level services that rely on these identities to function independently without requiring direct connection with Trusted Computing components.

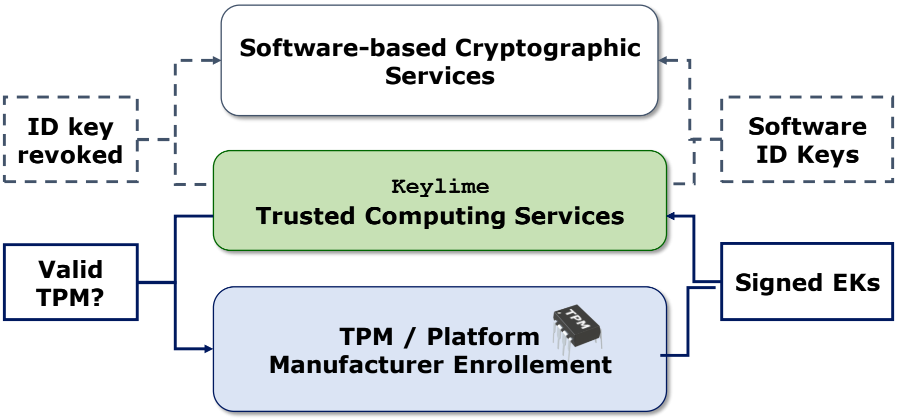

Software-based Cryptographic Services
ID key revoked
Software ID Keys
Keylime
Trusted Computing Services
Valid TPM?
Signed EKs
TPM / Platform Manufacturer Enrollement
TPM

Figure 5.1. Keylime layer between trusted hardware and software security services (Source [19]).

Figure 5.1 shows where the software Keylime layer is placed, decoupling the interaction between trusted hardware and high-level security services and supplying a plain interface accessible to developers and users with the recent security features.

5.2 Design

To overcome the limitations of existing methods, this framework combines trusted computing with IaaS to offer a hardware root-of-trust. This foundation permits tenants to establish confidence in the cloud provider's infrastructure and its systems. It leverages the TPM 2.0 chip and the IMA module as the RTM, assuming that attackers cannot physically access the platforms. It also assumes that the cloud provider is considered "*semi-trusted*", meaning it should enforce controls and policies to minimize the impact of possible attacks, particularly from malicious insiders, as the provider may have access to tenants' confidential data [1].

This section explains how trusted computing elements can be effectively utilized in a virtualized environment and describes a streamlined architecture for managing trusted computing services when cloud nodes are located on physical hosts. Keylime primarily comprises an agent, two server controllers, and a command-line interface. Each component operates independently and interacts through API-based communication. Figure 5.2 shows a simplified architecture highlighting connections encompassing the following Keylime's components.

- **Agent**: a service that operates on the target operating system for attestation, working with the TPM to register the Attestation Key and produce quotes. In other words, since it represents the attester machine, it provides information about its current integrity state by collecting necessary data for state verification and sending it as IRs. It is identified as a cloud agent in the network through a *Universally Unique Identifier (UUID)*. The Agent can track revocation alerts sent by the Verifier when an attestation fails, which is useful in environments where attested systems directly communicate, even if mutual trust is required. In these cases, a revocation message can suggest policy changes to stop the compromised system from accessing resources on other systems.

- **Registrar**: it manages the entire enrolment process by receiving register requests from the Agent. The Registrar implements the protocol for the Attestation Key Identity Certification by collecting the EK_{pub}, the EK certificate, and the AIK_{pub} from the Agent and proving an association proof that the retrieved AIK and EK are associated. The registration phase is described in section 5.4.1, and once registration is complete, the Agent will be ready for attestation enrollment.

- **Verifier**: is the central component of the Keylime architecture since it checks the integrity of a tenant's IaaS resources. It is responsible for conducting the Agent's attestation process and sending revocation alerts if the Agent leaves the *trusted state*. It interacts with the Registrar to obtain the AIK_{pub} key, essential for validating a TPM quote. This data typically includes a quote of PCRs, PCR values, the IMA log, and the UEFI event log.

- **Tenant**: represents the individual or organization utilising services across one or more cloud nodes and interacts with the Agent via a Command Line Interface or REST APIs. It works as an all-in-one platform for managing agents and supporting actions like adding or removing agents from the attestation process and checking the status of individual agents. Moreover, the Tenant can retrieve and use the EK certificate to verify the TPM's integrity and trustworthiness against a certificate repository. After the registration protocol, the Keylime framework kicks off, which provides the Agent with an encrypted payload specifying which service will be deployed securely. This starts the Three Party Bootstrap Key Derivation Protocol, explained better in section 5.4.2. A tenant can ask for information regarding the attestation of its running cloud services to the Verifier.

- **Software CA**: is a purely software-based certification authority designed to propagate trustworthiness from the TPM toward advanced security services and avoid each service trusted computing aware. When a cloud node receives a "revocation event", it uses and updates the Certificate Revocation List (CRL) to revoke the software identity key related to the untrusted cloud node.

- **Revocation Service**: it works with the Keylime Software CA by issuing a revocation event whenever a cloud node is detected untrusted by the Verifier. It is essential to inform all nodes registered for those services about the revocation. This notification process requires support from Software CA utilizing a CRL.

The primary function of Keylime is to shield tenants' sensitive data from the cloud provider. This data might include software identity keys or configuration details, such as a cloud-init script

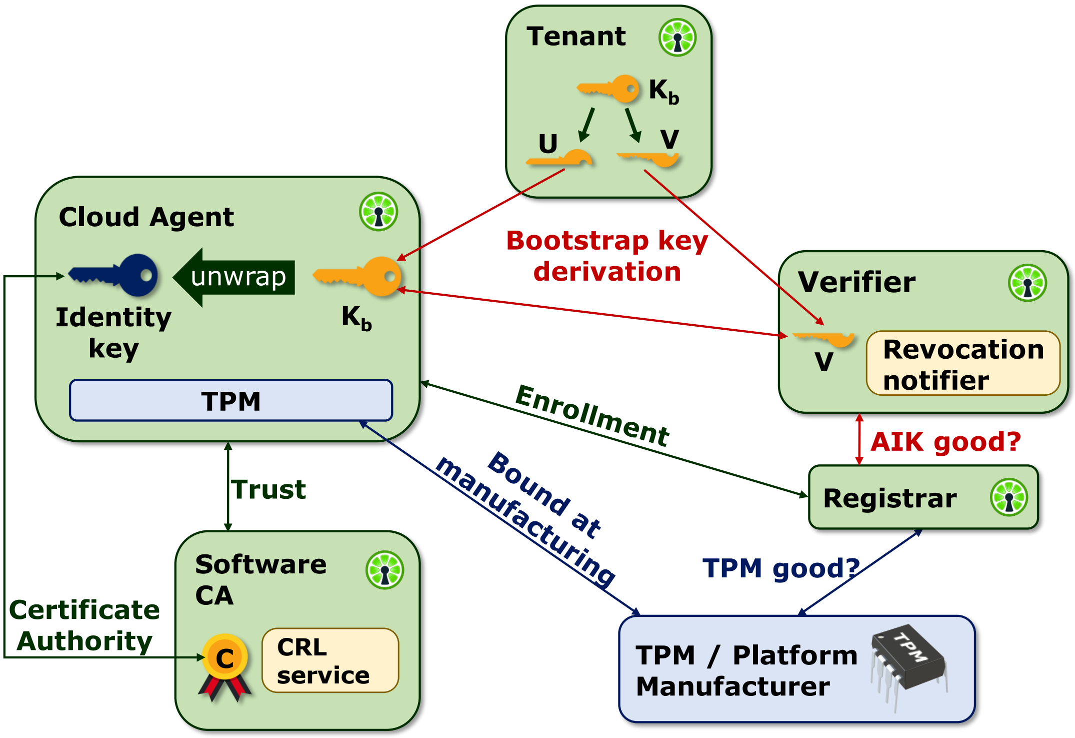


Figure 5.2. Keylime simplified architecture (source: [19])

(a standard mechanism to specify bootstrapping data, including secrets, for the cloud nodes). To achieve this, the Tenant begins by generating a new symmetric key K_b and uses it with the AES-GCM algorithm to encrypt sensitive data d obtaining Enc_{K_b}(d). Next, the Tenant requests the IaaS provider to create a new resource (physical or virtual node) and includes the encrypted result in the resource setup process. Once created, the provider returns the UUID and IP address to access the new node on the network.

5.3 Main Features

- **User Selected PCR Monitoring**: with Keylime's *TPM policy* feature, users can set up monitoring of a remote machine for any specified Platform Configuration Register (PCR) and related TPM quotes. This allows tracking specific changes in the system's measured state, offering targeted oversight of security-critical configurations.

- **Measured Boot**: In large-scale production environments, each node includes a TPM 2.0, which specifies that certain PCRs (i.e. PCRs 0-9) record boot-related measurements, like the UEFI BIOS and boot device firmware. However, comparing these PCR values to a single "golden" standard is impractical, as variations often arise. To overcome this limitation, modern UEFI firmware provides an event log through an ACPI table, and Linux kernels expose a detailed boot log file at `/sys/kernel/security/tpm0/binary_bios_measurements`. When paired with secure boot, this setup fully populates relevant PCRs, covering all boot components up to the kernel. Moreover, tpm2-tools Linux commands can process these logs to reconstruct the contents of PCRs [0-9] (or even [11-14]), permitting more precise boot measurement monitoring even in complex environments.

  Keylime leverages this new capability by enabling tenants to set a "measured boot reference state" (known as `mb_refstate`), which represents a specific policy for an agent node and is used for validating the Measured Boot ML. The Keylime verifier can then assess the node's boot event log against the received one, which can include both reference data for the node's expected boot events and rules for evaluating the log.

- **Runtime Integrity Monitoring**: Keylime's runtime integrity monitoring requires setting up Linux IMA (Integrity Measurement Architecture). Keylime enables policy creation and verification to ensure the integrity of an agent's files. It typically permits the creation of a list known as a *whitelist* to validate the IMA measurement log (ML). This whitelist includes trusted digests for configuration files and programs running on the cloud node. If any files are altered or an unauthorized key is added to a keyring, the hashes will not match, and Keylime will mark the Agent as compromised. It also supports IMA file signature verification to detect changes in immutable files. If any invalid signatures are found in the IMA log, a system reboot is needed to reset the log, allowing the Keylime Verifier to check the system's integrity accurately.

- **Secure Payloads**: Secure payloads in Keylime allow encrypted data (i.e. keys, passwords, root certificates) to be securely provided to a node. This encrypted data helps bootstrap the system with high confidentiality and integrity. The Keylime Tenant CLI (or REST API) sends the payload to the Keylime Agent, along with a key share called the *u_key* or user key. After the Agent meets all enrolment requirements (like TPM policy or IMA allow-list compliance), the Keylime Verifier sends the remaining key share, the *v_key* or verification key, allowing the Agent to decrypt the payload. Keylime supports two methods for secure payloads: Single File Encryption and Certificate Package Code. The first allows to specify the file by command line (in the `keylime_tenant` command) with the option `-f`. It will be encrypted with the bootstrap key and securely delivered to the Agent. The second automates many common actions tenants want to take when provisioning their Agents. This last mode natively supports certificate revocation by detecting an Agent that no longer satisfies its integrity policy and, in this case, creates a signed revocation notification [20].

5.4 Operational Phases

The following phases [19] summarise the entire Keylime framework behaviour:

1. *Physical Node Registration Protocol*
2. *Three Party Bootstrap Key Derivation Protocol*
3. *Continuous Remote Attestation*

5.4.1 Physical Node Registration Protocol

This phase starts the *Registration Protocol* through which AIK keys are generated and validated using TCG standards known by the Keylime framework. To validate the AIKs in the Registrar, Keylime implements a TPM-compatible enrolment protocol that starts interaction between the Agent and the Registrar. In this phase, an agent asks for the TPM's credential enrolment in the system. As shown in the figure 5.3, first, the node sends its UUID together with the $\mathrm{AIK}_{\mathrm{pub}}$, the $\mathrm{EK}_{\mathrm{pub}}$ and the EK certificate held by TPM. When the Registrar receives the credentials, it verifies the TPM EK's validity with the issued TPM manufacturer and temporarily saves the credential concern public area of the TPM object[5]. Since the AIK is not fixed to the platform (so the TPM object is not tied to a specific platform), it ensures privacy, but this phase must be repeated after each reboot. Here, the Registrar challenges the cloud node by proving that the received credentials are associated with the private counterparts. To do this, it sends a challenge taking an ephemeral symmetric key K_e, the digest computed over the TPM object public area and encrypt both with the $\mathrm{EK}_{\mathrm{pub}}$. The cloud agent once receives the blob, uses the `ActivateIdentity` TPM command to decrypt K_{e}, which only succeeds if the TPM holds both $\mathrm{EK}_{\mathrm{priv}}$ related to $\mathrm{EK}_{\mathrm{pub}}$ and $\mathrm{AIK}_{\mathrm{priv}}$ related to $\mathrm{AIK}_{\mathrm{pub}}$. In summary, it must demonstrate to the Registrar as proof of possession that it is the entity holding the right credentials. Indeed, it takes the decrypted K_{e} and uses it to compute the *keyed-hash message authentication code (HMAC)* over its UUID, sending it back and confirming its trustworthiness. The Registrar verifies this response by recalculating the HMAC. If the result is aligned with the local one, it marks the Agent as `active` and makes available the TPM credentials of the cloud node on demand, saving it on the persistent memory.

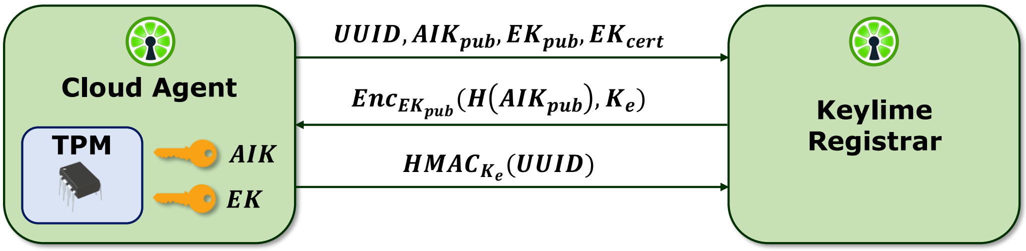

Cloud Agent
TPM
AIK
EK
$UUID, AIK_{pub}, EK_{pub}, EK_{cert}$
$Enc_{EK_{pub}}(H(AIK_{pub}), K_e)$
$HMAC_{K_e}(UUID)$
Keylime Registrar

Figure 5.3. Physical node registration protocol (source: [19])

5.4.2 Three Party Bootstrap Key Derivation Protocol

After completing the Node Registration Protocol, it became essential to securely provide the bootstrap key (K_b) to the cloud node as shown in the figure 5.2. The Three-Party Bootstrap Key Derivation Protocol comes into play to demonstrate that the Tenant would retrieve the bootstrap key K_{b} and that the cloud node will get the same information in a trusted state.

The protocol achieves these objectives by splitting the K_{b} between the Tenant and the potential cloud verifier. In particular, the Tenant divides the key K_{b} into two parts, U and V, in a more challenging reconstructing way. It takes a random value V, of the same length as $\mathrm{K_b}$ (typically 256-bit values) and obtains the second part U by computing the 5.1 formula. The Tenant then sends U to the cloud node to indicate its intent to derive $\mathrm{K_b}$. Meanwhile, it shares V with the Verifier, which will only release V to the cloud node after confirming its trusted status.

$$U = K_b \oplus V \tag{5.1}$$

Figure 5.4 shows the entire protocol, dividing it into phases A, B and C in order and focusing on all the interactions and exchanged values.

The protocol starts with the A phase, in which the Tenant contacts the Verifier over a protected channel, informing it that a new cloud agent is present and supplying specific data. These data refer to the cloud agent's UUID, the V part of the bootstrap key, the *IP address* and the *port* through which the Agent is reachable on the network, the TPM policy, the whitelist and a not required *Measured Boot refstate.*

The completion of the A phase triggers the parallel execution of the other two phases, B and C. This way, the Agent can independently receive the V-shared part from the Verifier and the U part from the Tenant.

Phase B involves the interaction between the Verifier and the Agent. The Verifier sends the Agent a freshly generated nonce ($\mathrm{nonce_{Verifier}}$). Along with the nonce, it also sends a PCR_{mask}, specifying the PCRs the quotes refer to. Since the Agent does not have a certified software identity key for secure communication, it creates a temporary asymmetric key, *NK*, and returns the newly generated public portion to the counterpart. Along with NK_{pub}, it sends a $Quote_{AIK}(nonce_{verifier}, 16\text{: } H(NK_{pub})$, $\mathrm{x_i{:}y_i}$). Here, 16: $\mathrm{H(NK_{pub})}$ refers to PCR 16, which contains the hash of $\mathrm{NK_{pub}}$, and will be compared upon quote validation with the received one in cleartext. Instead, $\mathrm{x_i}$ represents the multiple PCR requested by the Verifier and its respective value $\mathrm{y_i}$. However, it is impossible to check the authenticity of the received quote without the $\mathrm{AIK_{pub}}$. So, the Verifier requests the Registrar the missing information along with other trusted node's TPM credentials ($\mathrm{EK_{pub}}$ and $\mathrm{EK_{cert}}$) and then check the quote. Then, if the quote is valid, the Verifier compares the Tenant's TPM policy with the PCR values in the IR and validates the whitelist against the IMA ML, ensuring cloud node trustworthiness. If all verification steps are successful, the Verifier sends the V-shared part protected with the $\mathrm{NK_{pub}}$, which the Agent can decrypt and proceed to the "Continuous Remote Attestation" phase (explained in 5.4.3). If verification fails, the Verifier does not send the V share and marks the cloud node's state as `INVALID_QUOTE`. If phase C occurs before, so the Tenant has already shared the U value, there is no risk of exposure because the bootstrap key is unique for each cloud node.

Phase C is initially similar to the previous, except for some details. Practically, it sends obviously a different fresh nonce ($\mathrm{nonce_{Tenant}}$) but without $\mathrm{PCR_{mask}}$ because the Tenant is not interested in verifying the node's trusted state, so the quote truthfulness, but rather the authenticity of the TPM. Indeed, it first verifies the $\mathrm{NK_{pub}}$ correctness and then asks for the same credentials as the Verifier to perform other kinds of control. In particular, it checks if the public key in $\mathrm{EK_{cert}}$ matches the received $\mathrm{EK_{pub}}$, checks if the $\mathrm{EK_{cert}}$ signature is authentic and if its issuer is trusted. If any checks fail, the Tenant does not send the U share to the cloud agent and informs the Verifier, which marks the cloud node's state as `TENANT_FAILED` and halts the continuous attestation on the cloud node. However, if the TPM is authentic, the Tenant verifies the quote with the $\mathrm{AIK_{pub}}$ key and sends the U-share part encrypted with $\mathrm{NK_{pub}}$. Then, compute HMAC over the node's UUID with the starting $\mathrm{K_b}$ and the encrypted payload (d) with the same key.

Once the cloud agent receives either the U share from the Tenant or the V share from the Verifier, it computes the 5.2 formula to retrieve $\mathrm{K_b}$ and compares the received HMAC value by the Tenant with the local computed one using the reconstructed key.

$$K_b = U \oplus V \tag{5.2}$$

If the values match, the cloud agent again uses $\mathrm{K_b}$ to decrypt the encrypted payload and start the Tenant's service. After decryption, the cloud agent deletes $\mathrm{K_b}$ and V values while storing

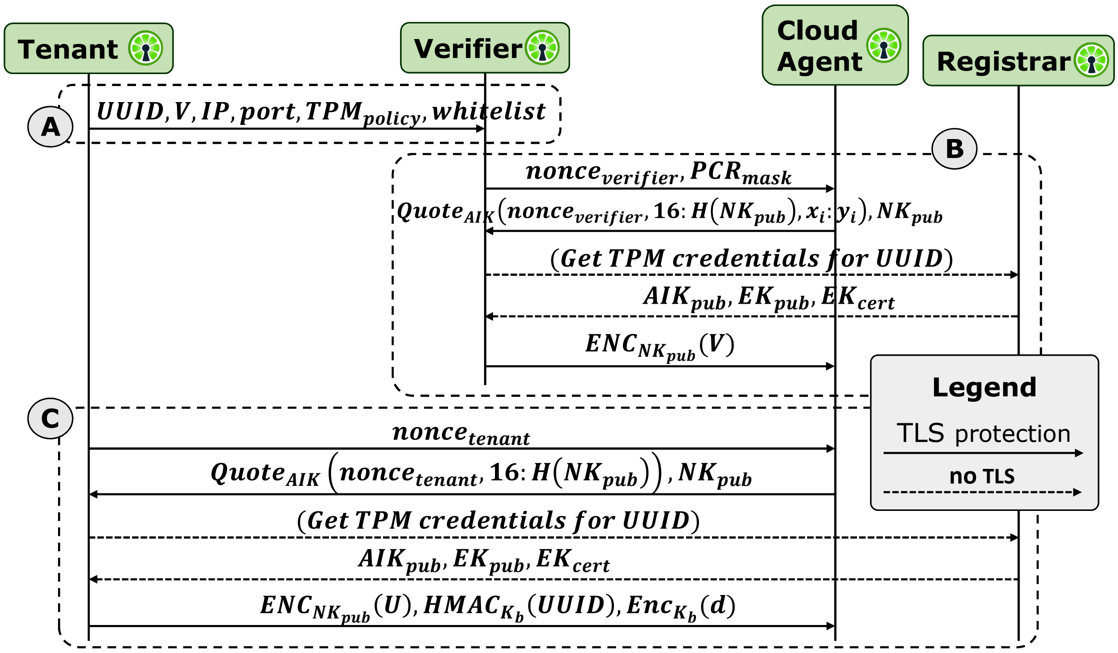

Tenant
Verifier
Cloud Agent
Registrar
A
$UUID, V, IP, port, TPM_{policy}, whitelist$
B
$nonce_{verifier}, PCR_{mask}$
$Quote_{AIK}(nonce_{verifier}, 16: H(NK_{pub}), x_i: y_i), NK_{pub}$
$(Get\ TPM\ credentials\ for\ UUID)$
$AIK_{pub}, EK_{pub}, EK_{cert}$
$ENC_{NK_{pub}}(V)$
C
$nonce_{tenant}$
$Quote_{AIK}\left(nonce_{tenant}, 16: H(NK_{pub})\right), NK_{pub}$
$(Get\ TPM\ credentials\ for\ UUID)$
$AIK_{pub}, EK_{pub}, EK_{cert}$
$ENC_{NK_{pub}}(U), HMAC_{K_b}(UUID), Enc_{K_b}(d)$
Legend
TLS protection
no TLS

Figure 5.4. Three Party Bootstrap Key Derivation Protocol (source: [19]).

U in the TPM's NVRAM for future use without requiring further interaction tenant-side. If a reboot or migration occurs, the cloud agent sends again a new $\mathrm{NK}_{\mathrm{pub}}$ with the TPM quote to the Verifier, which then sends the V-shared part back to the counterpart. The cloud agent can then recombine K_{b} and decrypt the payload again.

5.4.3 Continuous Remote Attestation

Upon finishing the Three Party Bootstrap Key Derivation Protocol, the framework starts the Continuous Remote Attestation phase. This phase enforces periodic attestation and ensures that, after confirming a trusted boot for secure services launched on the node and after the payload is delivered and decrypted, the Verifier continually monitors the cloud node's integrity over time. As illustrated in Figure 5.5, the Keylime verifier begins continuously polling the Agent to obtain current integrity reports based on IMA measurements of applications running on the node. This process ensures the system's ongoing trustworthiness. The Verifier reviews each new IR to detect changes in the system's integrity status.

For each report, the Verifier will verify the authenticity and validity of the received quote and whether the PCRs contain the expected values every time. In more detail, it checks if the signature of the quote is acceptable, validating it with the $\mathrm{AIK}_{\mathrm{pub}}$ retrieved from the Registrar. Then check the quote freshness using the provided nonce and if the quote refers to all specified PCRs in the $\mathrm{PCR}_{\mathrm{mask}}$. Moreover, verify the $\mathrm{NK}_{\mathrm{pub}}$ correctness sent by the Agent by comparing its digest with the PCR 16 value in the quote. Also, following the IMA ML verification process, check if the log matches the PCR 10 value and if the measurement events refer to those specified in the whitelist. In the end, check if the PCR values specified in the Measured Boot ML match the corresponding PCR in the IR and if this log match also the optional `mb_refstate` content file supplied by the Tenant.

The interposed time between consecutive attestation requests defines how quickly a potential compromise is detected. This interval is typically set to two seconds, though it can be adjusted in the Keylime configuration file. The minimum latency is around 500 milliseconds and can not be overtaken because TPM quote generation usually needs additional time.

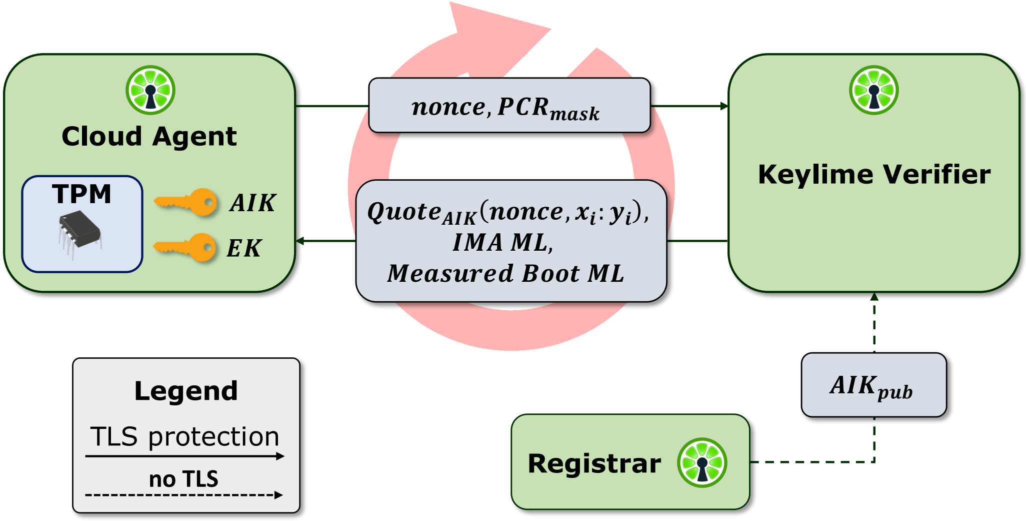


Figure 5.5. Continuous Remote Attestation

Revocation Framework

If the Verifier finds a cloud node untrusted, the Keylime verifier activates the *Revocation Framework*. This framework uses a service that leverages a *publish/subscriber* paradigm, through which the Verifier can publish a new *revocation event* to the service and forward the event to all the subscribers. In particular, a subscriber can represent the software CA that receives a revocation message from a potential untrusted node. This CA can then revoke the certificate of the identity key owned by that node and publish an updated CRL. A subscriber can also be a trusted cloud agent by executing a specific script to isolate an untrusted node upon receiving a new revocation message. It can also represent additional tools that must stay informed about events concerning the trusted computing layer.

Chapter 6

Related works

The rapid advancement of computing systems, especially with the rise of application container technologies, has brought new challenges to ensuring system integrity and security. The IMA architecture enables systems to verify their integrity through runtime measurements and attestation processes. Although the IMA architecture provides a foundation for addressing integrity-related challenges, its standard implementation encounters limitations when extended to accommodate more dynamic environments, such as cloud computing and containerised applications.

The core components of container technologies, such as images, registries, orchestrators, containers, and host operating systems, present various security risks [21]. Image risks include vulnerabilities from outdated images and malicious or untrusted content; registries instead, as centralised sources for images, face threats like insecure connections and stale or vulnerable images. Orchestrators, which manage the execution environment, are exposed to risks from improper access controls, unauthorised data access, and poor workload and network configurations. Containers themselves are vulnerable to runtime exploits, misconfigurations, and privilege escalation. In contrast, the host OS, foundational to container operations, suffers from a broad attack surface and dependencies on secure kernel components.

This chapter presents different non-standard solutions that extend the IMA architecture by introducing customisation for the attestation framework, designed to better accommodate the challenges of contemporary potential threats. These solutions aim to resolve critical security and privacy challenges in multi-tenant environments, particularly ensuring transparency in identifying application executions to specific tenants and mitigating the expansion of the attack surface. To address the increasing complexity of these environments, the proposed extension improves IMA's ability to manage the virtual and dynamic nature of containers.

The chapter aims to demonstrate the feasibility and effectiveness of extending IMA to implement mechanisms for real-time integrity measurement of container images and runtime behaviours. The security demands of modern application container technologies contribute to developing more secure and trustworthy computing environments and require a more scalable attestation process suitable for microservice architectures or orchestrated container deployments.

6.1 Container-IMA

Container-IMA [22] represent one of the first solutions of container attestation, developed by a research group at Peking University in 2019. It focuses on addressing privacy issues in container attestation, which are the central topic of discussion in this thesis due to the multi-tenant composition of cloud nodes. These issues can arise during the conventional remote attestation procedure. Indeed, attestation for physical nodes grants the verifier access to the entire measurement log of the attester, even if the purpose is to validate only the containers belonging to a specific tenant. With these conditions at the verifier level, an attacker can exploit the attester's vulnerabilities and potentially access container data from other tenants.

To address this, Container-IMA introduces a modified Linux IMA module that enables the partitioning of *Measurement Log (ML)* entries based on the specific container that produced them. This work's key contributions include enabling the attestation of a specific container while safeguarding the privacy of other containers and the host, preventing unnecessary exposure of details to a potentially malicious verifier. It also ensures the collection of integrity evidence for all components and dependencies of the targeted container using a *container-based PCR (cPCR)* secured by a hardware-based RoT, making any tampering detectable and obtaining robust protection. Furthermore, the approach minimises container attestation latency, ensuring that the overhead introduced by Container-IMA is practically negligible.

6.1.1 Architecture

As described in the previous chapters, IMA checks if a system is running by verifying that the software hasn't been tampered with. The approach leverages a verifier to validate a *prover*'s trustworthiness (or attester's) during the Remote Attestation procedure by providing evidence of its integrity, including a list of all the software running on it (the Measurement Log). The ML contains a detailed record of measurement values and essential metadata for software components, reflecting the platform's integrity status.

In a containerised environment, there are some open issues that this architecture aims to solve. The main one is that sharing the full ML can be a privacy concern, for instance, when multiple verifiers access data for different users on the same host. Typically, a Container management service, such as Docker Daemon, handles the creation and management of containers and all components running on the host system are categorised as host applications. When a client requests that an application be executed, the container management services locate and load the specified image, instantiating a container. Here, a verifier is a remote user interested in the integrity of their containers on the prover. If the verifier owns only a specific container, it should not have access to unauthorised details about other containers or the underlying host.

Fortunately, the Linux namespaces help protect privacy. This mechanism creates isolated environments for containers by partitioning system resources, such as mount points, hostnames, IPC, PID, and network, into separate instances across domains. Each container is typically assigned a unique namespace, ensuring isolation from other containers.

Basically, the trusted boot measures components during the platform's boot process. A measure-before-loading approach establishes a chain of trust that extends from the RoT to the OS kernel and so on to another level. Once trust reaches the application layer, IMA measures all software components as they load and records them in sequential order in the ML. This process integrates the loading sequence of software components into the chain at the application layer. So, The entire Chain of Trust can be partitioned into:

- **Integrity of the Prover's Boot Process** ($I_{\text{boot}}^{\text{Pro}}$): covers the period from powering on the prover to successfully loading the OS kernel, including components such as the BIOS, GRUB, and OS kernel;
- **Integrity of Containers' Dependencies** ($I_{\text{dep}}^{\text{Con}}$): relates to the container management services and the files or libraries they rely on;
- **Integrity of a Container's Boot Process** ($I_{\text{boot}}^{\text{Con}}$): includes the container images and boot configurations used when the container management services launch a container;
- **Integrity of a Container's Applications** ($I_{\text{app}}^{\text{Con}}$): spans from the successful launch of a container by the container management services to its shutdown, encompassing all components running within the container;
- **Integrity of Host Applications** ($I_{\text{app}}^{\text{Host}}$): extends from the successful loading of the OS kernel to the shutdown of the prover, excluding the container management services and all containers;

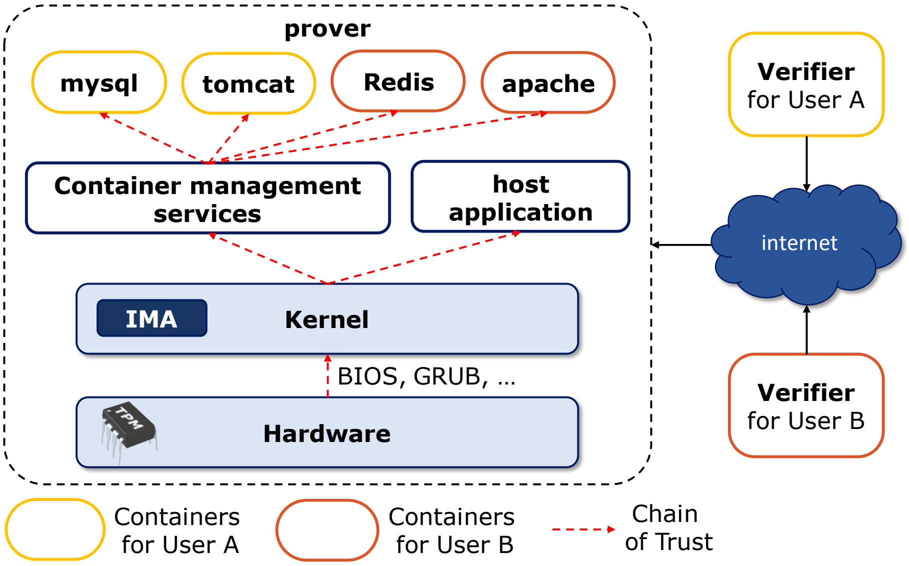


Figure 6.1. Use Case in a Container Setting (Source [22]).

Figure 6.1 shows a scenario where a prover hosts multiple containers for different users, emphasising the chain of trust specific to containers. In this case, there are several containerised applications, some for user A and some for user B. For example, user A wants to ensure that their container (e.g., MySQL) is operating correctly, such as booting from the correct image and loading trusted software. Containers rely directly on the container management service and, due to namespaces, are isolated from other host applications.

As a result, the chain of trust for each container consists of specific components: $I^{\text{Pro}}_{\text{boot}}$, $I^{\text{Con}}_{\text{dep}}$, $I^{\text{Con}}_{\text{boot}}$, and $I^{\text{Con}}_{\text{app}}$ (with the last two referred to the container under examination). When a verifier requests attestation data for a container, the prover can only provide these specific subsets of information. Any other measurements not related to these subsets, particularly those about $I^{\text{Host}}_{\text{app}}$ or other containers' $I^{\text{Con}}_{\text{app}}$ and $I^{\text{Con}}_{\text{boot}}$, must not be disclosed to the verifier.

6.1.2 New components

This previous solution meets specific privacy conditions, but additionally, a strong requirement exists: the verifier cannot determine whether its container is the only one running on the prover's system. Container-IMA divides the traditional ML into the mentioned partitions, and to implement them, it modifies the IMA module by incorporating three additional components:

1. the *Split Hook* component: lightweight virtualisation relies on namespaces to isolate system resources, ensuring that each container is assigned to a separate namespace. The Split Hook monitors system calls related to namespace generation and informs the kernel about events that require ML partitioning.

2. the *Namespace Parser* component: identifies the namespace number of the process that generates an event to be measured. Associating events with their respective namespace numbers allows events to be assigned to the correct ML partition, corresponding to the container that triggered the event.

3. the *container-based PCR Module*: is a kernel-level data structure designed to emulate the host's TPM PCRs, safeguarding each ML partition. Each cPCR contains three key fields: the value derived from applying the `extend` operation to the container applications partition, the namespace number, and a secret. The secret ensures the verifier cannot access data from other tenants' containers.

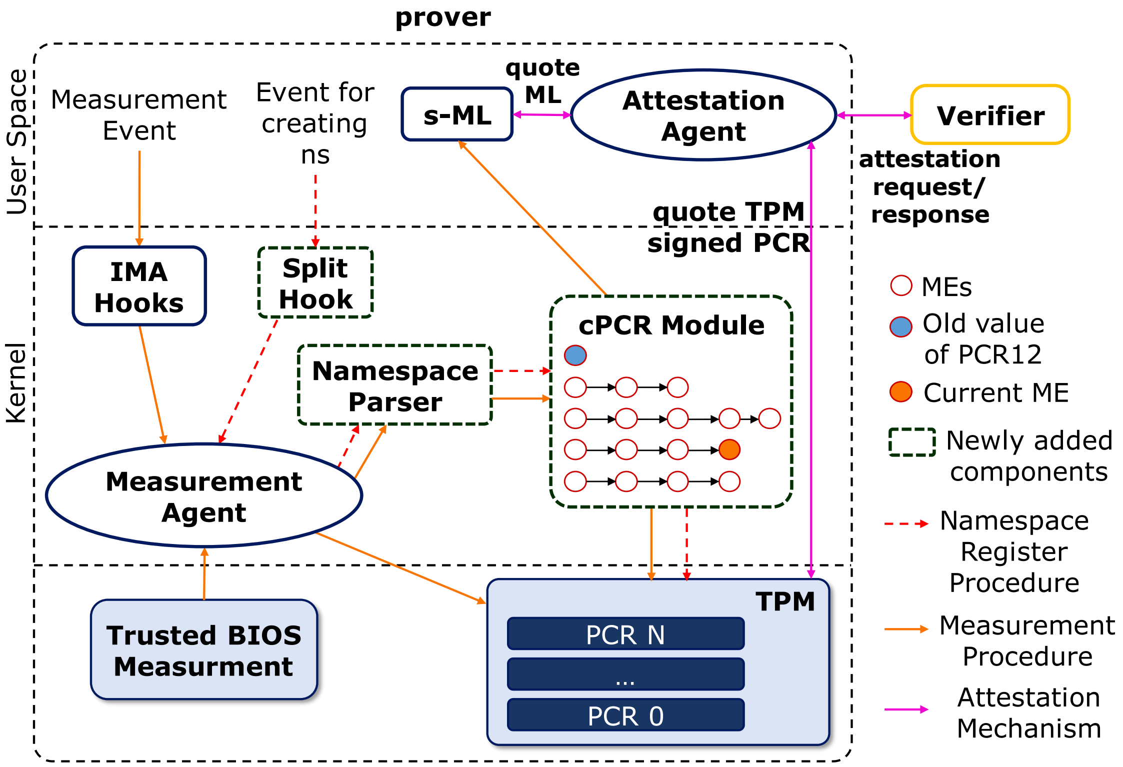


Figure 6.2. Complete Architecture of Container-IMA (Source [22]).

Figure 6.2 depicts the entire Container-IMA architecture, highlighting the newly added components described above. Note that a *Measurement Event (ME)* signifies an event that triggers a measurement. IMA determines MEs based on a defined policy. The complete architecture enabling individual container attestation relies on two main mechanisms: the *measurement mechanism* and the *attestation mechanism*.

6.1.3 Measurement Mechanism

This section explains the measurement mechanism, dividing it into two basic procedures: the *Namespace Register Procedure*, which enables the initial partitioning of the ML for a container based on its mount namespace and the *Measurement Procedure*, which assigns a specific ME to its corresponding ML partition. It is important to note that components involved during the prover's boot process ($I_{\text{boot}}^{\text{Pro}}$) are measured by the trusted boot and safeguarded by PCR0-7. Since MEs generated during the boot phase are already placed in a separate ML, they are excluded from the subsequent discussion.

Basic Namespace Register Procedure

The main issue is to differentiate between containers from the kernel's perspective. Since each container is expected to have a unique namespace, the kernel can distinguish containers by analysing the namespace number of a running process during the process. This enables the separation of $I_{\text{app}}^{\text{Host}}$ and $I_{\text{app}}^{\text{Con}}$.

The Namespace Register Procedure is designed to allow the kernel to partition the ML. In this architecture, the kernel maintains multiple doubly linked lists to represent the separate measurement log (s-ML) derived from the ML.

Following the figure 6.2, the Split Hook informs the IMA Measurement Agent when a new mount namespace needs to be generated. In response, the agent allocates new data structures to separate the MEs generated by processes in the new namespace. Container-IMA maintains a measurement list in kernel memory for each namespace, with each list having its own *separated Measurement Log (s-ML)*, which contains the MEs generated within that specific namespace *ns*:

$$s\text{-}MLs \ = \ \{< value, ns >\}_n \ = \ \{< \{measure(ME_{ns})\}, ns >\}_n \tag{6.1}$$

where n is the number of generated namespaces and s-MLs represent the set of all separated logs that can be represented. To safeguard the integrity of these s-MLs, the solution created a simulated set of PCRs known as container-based PCRs. If there are n partitions in the ML, the complete list of cPCRs can be represented as:

$$cPCR\text{-}list \ = \ \{cPCR\}_n \ = \ \{< value, ns, secret >\}_n \tag{6.2}$$

As mentioned, the cPCR is assigned a unique value, a namespace number, and a secret. Once the namespace is created, the agent instructs the Namespace Parser to extract the new namespace number and then forward it to the cPCR Module, which will request the TPM generate the secret. This random value created by the TPM obscures the cPCRs of other containers about other namespaces from a specific verifier. Ultimately, the module creates a new list of cPCR (and a corresponding s-ML) established for the newly created namespace:

$$cPCR\text{-}list_{new} \ := \ cPCR - list_{old} \ \cup \ \{< all\_zero, ns, secret >\} \tag{6.3}$$

$$s\text{-}MLs_{new} \ := \ s\text{-}MLs_{old} \ \cup \ \{< \{\}, ns >\} \tag{6.4}$$

Basic Measurement Procedure

The measurement procedure, instead, is in charge of measuring and logging MEs. When an ME is created, it activates the IMA hooks, which alert the Measurement Agent to measure the ME. The resulting measurement and the current process's namespace number, extracted by the Namespace Parser, are forwarded to the cPCR module. If the ME lacks a mount namespace or is not registered in the cPCR Module, it is treated as part of the `host applications`. In this case, the measurement result is extended into PCR10 and logged in the `ascii_runtime_measurements` file. Otherwise, if the ME has a mount namespace registered via the Namespace Register Procedure, the module identifies the appropriate target cPCR and s-ML on which the measure will be logged:

$$target\text{-}cPCR \ = \ \{cPCR \mid cPCR.ns \ == \ ns \ \cap \ cPCR \in cPCR - list\} \tag{6.5}$$

$$target\text{-}ml \ = \ \{s\text{-}ML \mid s\text{-}ML.ns \ == \ ns \ \cup \ s\text{-}ML \in s\text{-}MLs\} \tag{6.6}$$

Next, the cPCR module updates the target cPCR by extending it with the current ME (following the extend operation outlined in the TPM specifications: `PCR_Extend`) and adds the ME to the target s-ML:

$$target\text{-}cPCR.value_{new} \ := \ HASH(target\text{-}cPCR.value_{old}, ME.node_{hash}) \tag{6.7}$$

$$target\text{-}ml.value_{new} \ := \ target\text{-}ml.value_{old} \ \cup \ measure(ME) \tag{6.8}$$

The integrity of each s-ML is safeguarded by its corresponding `cPCR.value`, allowing a verifier to detect tampering by comparing the `cPCR.value` with the computed `PCR_Extend` operation over the s-ML entries. However, since cPCRs reside in kernel memory, a hardware-based RoT does not protect their integrity. For this reason, Container-IMA has implemented one way to permit the cPCR module to bind cPCRs into a physical PCR.

Bind cPCRs to Hardware-based RoT

After updating the target cPCR and adding the ME to the target s-ML, the cPCR module connects the cPCRs to a physical PCR, which in a TPM 1.2 is the PCR12, through a series of steps. First, the cPCR Module saves the current value of PCR12 in an internal variable named `historyPCR` used by verifiers to ensure the integrity of the cPCR list. Next, the cPCR Module calculates:

$$sendcPCR_i \ := \ cPCR_i.value \quad xor \quad cPCR_i.secret \tag{6.9}$$

for each cPCR by performing an XOR operation between the template hash value and the secret associated with that register. The system then stores the digest of all cPCRs. This operation is carried out because when the verifier receives the sendcPCRs list, it can only disclose the cPCR.values for the containers whose secrets it knows. The `cPCR.value` of other containers remains hidden, preventing a malicious verifier from deducing details about those containers.

Finally, the system calculates the value of PCR12 by extending its old value with a specific value: $tempPCR_n$. This value is the result of the extension operation performed over all the cPCRs considering each time the $\mathrm{sendcPCR_i}$ value and the previous one:

$$tempPCR_i \ := \ HASH(tempPCR_{i-1} \,||\, sendcPCR_i), \forall i \in [1, n] \tag{6.10}$$

where the first $\mathrm{tempPCR_0}$ value is initialised to $\mathrm{sendcPCR_0}$. In this way, the new value of PCR12 is computed after saving its old status into the `historyPCR` variable. So, a hardware-based RoT is created since TPM's PCR12 safeguards the integrity of the sendccPCRs list, and each sendccPCR ensures the integrity of its corresponding s-ML.

The verifier's role permits checking container integrity during attestation with these conditions. Firstly, it checks the signature received from the prover over the quote, containing the s-ML specific for that container, along with the `historyPCR`, all the sendcPCR values and evidence of the right PCR12 value. Note that the verifier could perform a reverse xor operation to obtain the cPCR.value from the sendcPCR if informed about the secret at container creation. Once the list is valid, it can extract the final cPCR.value by extending all the $\mathrm{cPCR_i}$.value, each from the $\mathrm{sendcPCR_i}$. If the final value matches the value calculated by extending all template hash in the received s-ML, it can also demonstrate the list integrity.

6.1.4 Attestation Mechanism

The Attestation Mechanism enables a remote verifier to attest to the integrity of a specific container and its dependencies based on traditional Remote Attestation. For Container-IMA, a cluster is assumed to have an effective user management system, such as Kubernetes [6], avoiding unauthorised verifiers from participating in the attestation mechanism. For this reason, verifier identification is not considered in this section.

As mentioned, the mechanism is based on the traditional RA approach where an authorised verifier sends a nonce to the *Attestation Agent* and the `containerID` of the container to attest. The nonce is a randomly generated number created by the verifier to avoid replay attacks, while the `containerID` refers to the UUID of the target container. Upon receiving the request, the Attestation Agent responds to the verifier with an IR containing the following details:

1. the TPM's *quote*, signed with the AIK containing the nonce and the related PCR values;
2. the `sendcPCR` value and the `historyPCR` of the PCR12;
3. the s-ML for prover's boot time ($I^{\mathrm{Pro}}_{\mathrm{boot}}$) ;
4. the s-ML for container's dependencies ($I^{\mathrm{Con}}_{\mathrm{dep}}$);

5. the s-ML for container's boot time ($I^{\text{Con}}_{\text{boot}}$), providing complete information for the target container while only including the template hash for the other ones;
6. the s-ML for the target container's applications ($I^{\text{Con}}_{\text{app}}$).

Verifier Workflow

Upon receiving the response from the Attestation Agent, the verifier first verifies the authenticity of the TPM quote by validating the signature using the public part of the AK. If the quote is authentic, the verifier checks the nonce matches the one sent in the IR request. This permit establishes the trustworthiness of the nonce and allows the verifier to confirm the response's freshness. Secondly, the integrity of the s-MLs for bootstrapping the prover ($I^{\text{Pro}}_{\text{boot}}$) and the target container ($I^{\text{Con}}_{\text{boot}}$) can be verified using the trusted PCR values. These s-MLs are stored in specific physical PCRs: PCR0-7 for ($I^{\text{Pro}}_{\text{boot}}$) and PCR11 for ($I^{\text{Con}}_{\text{boot}}$). Their authenticity is validated by simulating the `PCR_Extend` operation and matching the simulation results with the trusted PCR values. Thirdly, the trusted PCR12 is used to validate the authenticity of the received sendcPCRs. To do this, perform the `extend` operation with all $\text{sendcPCR}_{\text{i}}$ to obtain $\text{tempPCR}_{\text{n}}$ using equation 6.10. If the PCR12 doesn't match, it indicates an attack has happened. Consequently, the verifier uses the container's secret to obtain its cPCR.value and all-zero to obtain cPCR_0.value used to verify the integrity of the s-MLs for the container's applications ($I^{\text{Con}}_{\text{app}}$) and its dependencies ($I^{\text{Con}}_{\text{dep}}$), in order. If all the validation procedures mentioned above are successful, all related s-MLs are considered trusted. The verifier can then validate the s-MLs by comparing them to his expectations, aligning them with traditional Trusted Computing and collecting them from software and hardware manufacturers.

6.1.5 Consideration

The container-IMA architecture permits solving the issue of tenant privacy in the containerised scenario, avoiding overturning the container architecture, but it has some disadvantages. First, Container-IMA relies on the TPM 1.2 specification, which is outdated and no longer recommended for new deployments. This solution's main limitation is that it does not support containers in the same namespace, which is particularly problematic in Kubernetes environments. The starting point for improvement is enabling Container-IMA to work with containers that share namespaces, as Kubernetes often deploys applications across multiple containers within the same pod.

Container-IMA hides certain details, like cPCR values, to protect privacy. This is a pro since attackers cannot directly gain details about other containers but may still infer valuable information from patterns or side-channel attacks if the communication is not trusted. For instance, if an attacker manages to intercept the `cPCR.value` multiple times during a remote attestation process, can learn knowledge to determine which container is currently active. Moreover, by observing changes in the `historyPCR` over time, the attacker can deduce when a container has executed certain operations. Ensuring malicious attackers cannot infer meaningful data would enhance the system's robustness against sophisticated attacks.

6.2 Docker Integrity Verification Engin (DIVE)

DIVE[23] is a solution proposed by the TORSEC research group at the Polytechnic of Turin, specifically targeting the Docker container engine. It is designed to provide integrity evidence for lightweight cloud environments and operates at the compute node level, requiring only a single TPM per node. DIVE enables the integrity verification of the host, the container engine, and the running containers. Authenticating the integrity state of services within containers through the TPM facilitates verification during the Remote Attestation phase. A key advantage of DIVE is its ability to identify which specific container has been compromised. This permits the immediate termination and replacement of the affected container without resetting a full platform. As a result, DIVE enhances the efficiency of the Remote Attestation process, making it a practical and scalable solution for modern cloud environments. The schema consists of three main components:

- the **Attester**, which serves as the target of the Remote Attestation process, is equipped with a TPM and runs the IMA framework in addition to the Docker container engine; it also includes the RA agent, responsible for initiating the RA process by sending Integrity Reports and interacting with the Verifier;
- The **Verifier**, as the core of the architecture, evaluates the integrity state of each Attester and its containers;
- the **Infrastructure Manager** is responsible for provisioning containers based on user requirements and maintaining a record of the association between Attesters and containers, using their Universal Unique Identifiers (UUIDs) for identification.

When a cloud user requests the integrity of one of its services, the Infrastructure Manager initiates the Remote Attestation process. It sends the Verifier a list of containers and their respective host machines to be checked. The Verifier then contacts the specified Attesters, requesting integrity reports that include evidence for the host system and all running containers. Using a reference database, the Verifier validates the measurements of the targeted containers and generates an attestation report for the Infrastructure Manager. If a container is compromised, the Infrastructure Manager can terminate it and launch a new one without rebooting the system. However, a system reboot is required if the host operating system is compromised since a tamper Attester may provide false reports, making compromised containers undetectable.

6.2.1 Consideration

DIVE leverages TPM-based hardware RoT for container attestation with minimal performance impact. It permits the identification of compromised containers or hosts and enables efficient recovery. It requires no changes to containerized applications or Verifier implementation but only to the Attester machine, making it practical for real-world use. This solution, however, presents some drawbacks inherent to the TPM specification, such as the lack of a Device Mapper as a storage driver for Docker. Firstly, DIVE is based on the TPM 1.2 specifications for various reasons, which are, unfortunately, deprecated. Another limitation is the potential collisions for deviceID assigned to containers. In particular, if a container is stopped and a new one is started, the new container might obtain the same ID as the previously terminated container, causing attestation failures. Furthermore, DIVE does not protect against in-memory attacks, requiring separate user and kernel space protections.

6.3 Docker Container Attestation

Generally, the MEs generated by processes on the host system and its containers are stored together in the same ML. During the container attestation procedure, it is difficult to identify which MEs belong to a specific container rather than another one. In the recent past, the TORSEC research group at the Polytechnic of Turin has developed a valid solution for container attestation that aims to associate each event present in the ML with the container or host system that generates it. To do this, this work is based on TPM 2.0 specification and the concept of Control groups. Control groups (cgroups) are Linux features which limit, account and isolate the resource usage (i.e. CPU, memory, disk I/O) of a collection of processes, allowing the organisation of these into hierarchical groups.

In the Linux kernel, the Docker process creates a control group for every new container generated. This is to ensure the separation of processes. Containers function as processes, with the main application of each container acting as the *init process*. Consequently, all processes within a container are descendants of the init process and cannot change their groups. In summary, all processes within a container share the same groups. It has been established that Docker assigns a unique identifier to each container upon its creation. This identifier, referred to as the *container full-ID*, is a 256-bit random code corresponding to the name of the newly created control group associated with the container. The work presents a new version of the IMA template (`ima-dep-cgn`), with the consequent kernel function adjustments. It incorporates an additional template descriptor of the form "`dep|cgn|d-ng|n-ng`" adding the control group name and all the dependencies of the process in addition to the previous template. The incorporation enables a Verifier to bind an ML entry with his container, only if the same conditions are met. Firstly, the control group name field must be filled with a container-full-identifier. Secondly, a container runtime must be present in the dependencies list field. If either of these conditions is not met, the entry will be considered to belong to the host system. The RA framework used is Keylime, which required modifications to enable container registration and support for the new template.

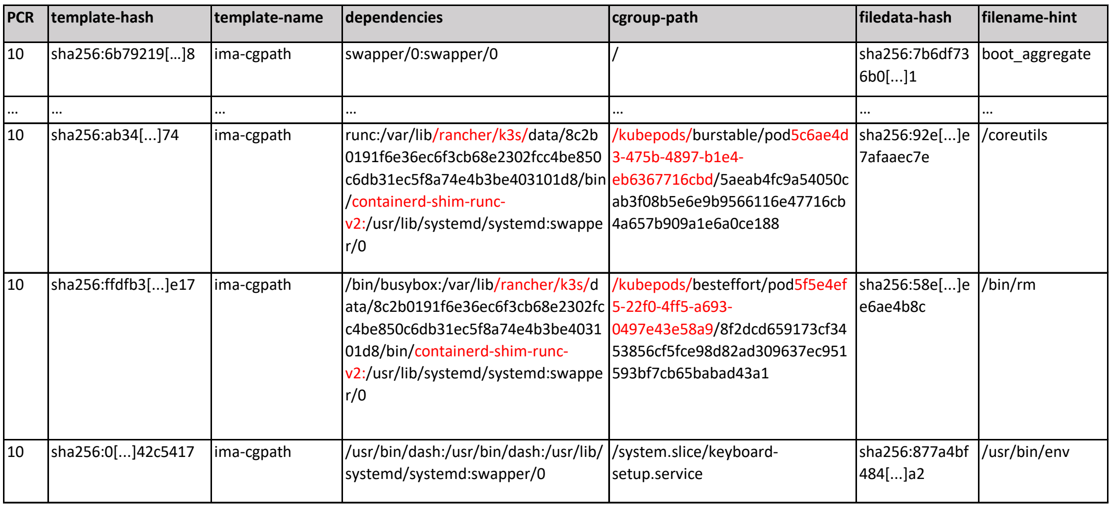
This work represents a valid container attestation solution, with the potential for adaptation and integration into Kubernetes, where the smallest scheduling unit is the pod, rather than a container.

6.4 Kubernetes Pod Attestation

More recently, the research group TORSEC of the Polytechnic of Turin has devised a method for directly attesting the pod [24]. This work, instead, would identify all entries in the ML related to a specific pod. To achieve this objective, the development of a revised IMA template (`ima-cgpath`) was imperative, with the incorporation of the pod identifier as a mandatory component within the attestation process. The RA framework utilised is Keylime, and modifications were necessary to facilitate pod registration and the integration of the proposed template. The solution first aims to discriminate the ML to understand the pods or host trustworthiness against an associated whitelist. Then, following the setup of the cloud environment and modification of Keylime, permit operations such as the periodic remote attestation and life cycle management.

IMA template

In the context of Kubernetes, a unique identifier, known as a UUID, is assigned to each object. This serves to differentiate between past instances of entities with similar characteristics. It has been observed that this UUID is consistently present in the control group path string associated with a particular pod that is managed by Kubernetes. This consistency can be attributed to the features of the control group in the Linux kernel, which is responsible for writing pod strings in a specific format. The new version of the IMA template designated as (`ima-cgpath`) facilitates the display of the specified UUID within a designated field in the ML, namely `cgroup-path`. Consequently, in the context of pod attestation, a Verifier is able to bind a select number of ML



PCR	template-hash	template-name	dependencies	cgroup-path	filedata-hash	filename-hint
10	sha256:6b79219[...]8	ima-cgpath	swapper/0:swapper/0	/	sha256:7b6df736b0[...]1	boot_aggregate
...	...	...	...	...	...	...
10	sha256:ab34[...]74	ima-cgpath	runc:/var/lib/rancher/k3s/data/8c2b0191f6e36ec6f3cb68e2302fcc4be850c6db31ec5f8a74e4b3be403101d8/bin/containerd-shim-runc-v2:/usr/lib/systemd/systemd:swapper/0	/kubepods/burstable/pod5c6ae4d3-475b-4897-b1e4-eb6367716cbd/5aeab4fc9a54050cab3f08b5e6e9b9566116e47716cb4a657b909a1e6a0ce188	sha256:92e[...]e7afaaec7e	/coreutils
10	sha256:ffdfb3[...]e17	ima-cgpath	/bin/busybox:/var/lib/rancher/k3s/data/8c2b0191f6e36ec6f3cb68e2302fcc4be850c6db31ec5f8a74e4b3be403101d8/bin/containerd-shim-runc-v2:/usr/lib/systemd/systemd:swapper/0	/kubepods/besteffort/pod5f5e4ef5-22f0-4ff5-a693-0497e43e58a9/8f2dcd659173cf3453856cf5fce98d82ad309637ec951593bf7cb65babad43a1	sha256:58e[...]ee6ae4b8c	/bin/rm
10	sha256:0[...]42c5417	ima-cgpath	/usr/bin/dash:/usr/bin/dash:/usr/lib/systemd/systemd:swapper/0	/system.slice/keyboard-setup.service	sha256:877a4bf484[...]a2	/usr/bin/env

Figure 6.3. Example of ML with ima-cgpath template.

entries to a specific pod, provided that the format of the `cgroup-path` field corresponds to the specifications outlined by the control group when handling Kubernetes pods. The format of this field entry remains constant in order sense (string - UUID - string), as illustrated in Figure 6.3. However, it may vary in terms of keywords or the addition of the container ID to the associated container. The functionality of the dependencies field enables direct determination of the entry's association with the host or a container, as indicated by the presence of the container runtime string.

Keylime Integration

The solution is based on the Keylime Remote Attestation framework with some modifications. In more detail, it would integrate the Keylime framework in the Kubernetes environment, with a specific focus on ensuring the integrity of the nodes and the pods running within them. The whole architecture, shown in figure 6.4, helps to understand that the integration involves the strategic placement of Keylime components: the Verifier, Tenant, and Registrar operate within the Kubernetes Master Node, instead the Keylime Agent runs on the Worker Nodes. The integration modifies Keylime's execution flow, including the registration of pods and hosts, which is managed by the Keylime Tenant component.

Pod Registration

The essential element of this solution is the *pod registration*, which is situated after the host registration with the Keylime Verifier. This helps the registration of pods that operate on the node, thereby ensuring that only the anticipated ones are running. It is imperative to note that each pod is initially assigned a START state and can subsequently become either TRUSTED or UNTRUSTED by the Verifier. The unique identifier (UUID) is retrieved from the control group path field to check against a list of registered pods. If the pod (through the UUID) is

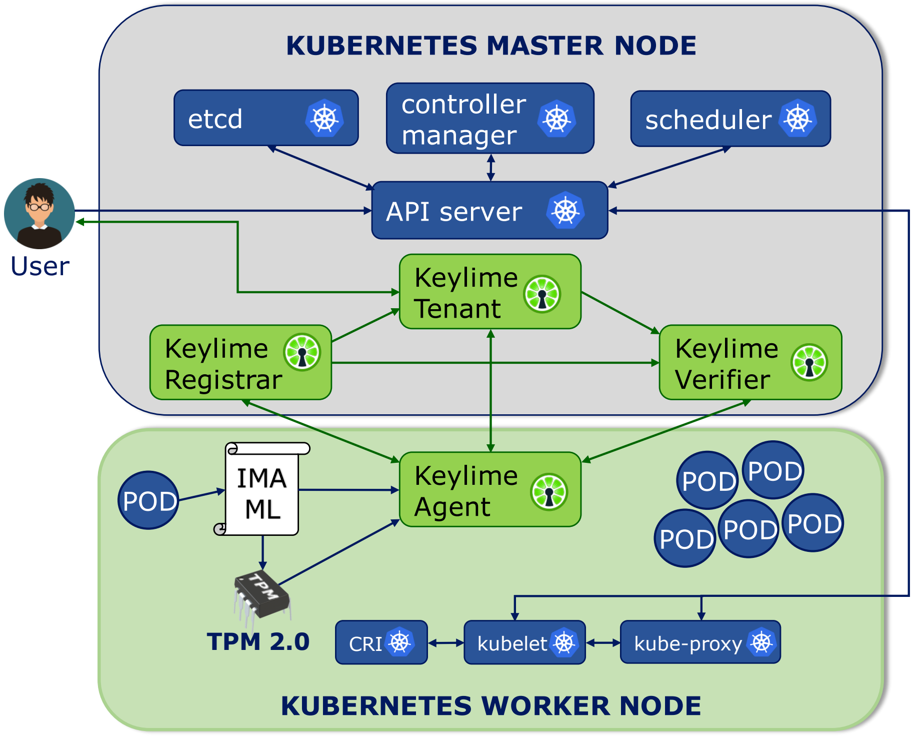


Figure 6.4. Architecture for Kubernetes Pod Attestation solution.

not registered, the host is deemed untrusted, and validation stops. If it is registered, the pod's measures are evaluated against a whitelist containing the golden values used to validate ML entries. If a registered pod is found to be untrusted, it will be isolated and replaced with a fresh version, but with a different UUID. Even though the pod can be considered untrusted, the host is considered trusted in order to prevent the entire platform from restarting.

Remote attestation

The Keylime Verifier validates the integrity of both the host and its pods. This happens after establishing the integrity of the Measurement List (ML) against PCR 10. The user triggers the remote attestation by sending a request to the Keylime Tenant, which in turn registers the Agent with the Keylime Verifier, although there is direct communication between them.

6.4.1 Consideration

The main aim of this solution is to protect the whole Kubernetes environment, including its nodes and pods. By checking pods, the system can spot any unauthorised or unexpected processes running on worker nodes. This method makes sure that everything is secure, but it also creates a strong base for adapting to realistic situations in the future.

In the interest of simplicity, the current implementation of the Keylime system consolidates all its components on the master node. Nevertheless, in a more general and complex scenario, it is possible that multiple verifiers could be located outside the master. The presence of multiple verifiers introduces potential privacy concerns, as a verifier, upon receiving a list of measurements for validation, might gain access to information about pods that is not relevant to its purpose. Moreover, the filtering measurement log is deemed to be trivial if the standard template hash fields are detectable with offline guessing. It is, therefore, necessary to establish a method of rendering the ML unpredictable and then performing a filtering process based on the pod identifier tied to the verifier's tasks.

Chapter 7

Proposed solution - Design

This chapter presents the design of the proposed solution, addressing key challenges related to privacy and security in remote attestation for Kubernetes environments. The chapter commences with an outline of the problems addressed, focusing on the risks associated with multiple verifiers accessing sensitive information in Measurement Logs and the necessity for a privacy-preserving approach. The proposed solution is then introduced, ensuring controlled access to attestation data while maintaining system integrity. To enhance security, the chapter details modifications to the IMA, including changes to the logging mechanism to prevent offline guessing attacks. The architecture of the proposed solution is then described, highlighting the interactions between different components within the cluster. The subsequent examination of the registration process involves a thorough discussion of the roles of Worker nodes, the Master node, and verifiers, along with the authorization mechanisms that delineate the verifiers' authority to attest to specific pods. The subsequent section details the verification process, explaining how integrity checks are performed while minimizing data exposure. The chapter culminates in a privacy and guessing analysis, evaluating how the proposed modifications enhance security and prevent unauthorised access to sensitive information. This structured approach ensures that the solution is both effective and scalable for large, multi-tenant cloud environments.

7.1 Problems addressed

In large-scale cloud environments like Kubernetes, privacy concerns arise when multiple verifiers access Measurement Logs to assess node and pod integrity. These logs may inadvertently expose sensitive information during the attestation phase, including data about unrelated pods or workloads, risking confidentiality breaches. The challenge becomes more significant as the system scales, as external verifiers often lack full context about the environment, making it difficult to enforce strict access controls. This can lead to verifiers accessing information beyond their scope without proper mitigation, undermining trust and privacy in multi-tenant systems. Another key challenge during attestation is ensuring that Measurement Logs cannot be easily reverse-engineered. Filtering ML entries becomes ineffective if standard template hash fields, which often follow predictable patterns, can be guessed offline, potentially exposing details about existing measured components or configurations.

To overcome these challenges, this chapter introduces a privacy-preserving solution that ensures that verifiers only receive the specific attestation data relevant to their task, without exposing the broader contents of the Measurement Logs. A design that obscures the internal distribution of pods across Worker nodes is necessary, preventing verifiers from inferring sensitive deployment details or gaining unnecessary visibility into the cluster's structure. The solution should also make the Measurement Logs unpredictable, mitigating the risk of offline guessing attacks and can be achieved by incorporating dynamically generated metadata into the logs. This approach is expected to render the content of the ML entries significantly more opaque to external verifiers, thereby ensuring that only authorised details are discernible.

7.2 Proposed Solution

The proposed solution centres on the redefined attestation protocol within the Kubernetes environment and the introduction of a new IMA template to overcome certain open privacy issues. The objectives of this work are twofold: firstly, to enhance the efficiency of the remote attestation procedure, and secondly, to improve the privacy of sensitive information when it is shared. This approach ensures that verifiers do not directly access Worker nodes or their Measurement Logs; instead, all attestation requests and responses are routed via the Master node. In this role, the Master ensures that all communication and attestation requests pass through a centralized point, abstracting the cluster's underlying structure from the verifiers. This decoupling abstracts the cluster's internal details from the verifier, preventing it from gaining direct knowledge about pod distribution and geographical details across nodes. The Master forwards the filtered measurement data received from the Agent to the verifier, ensuring that only the specific measurements required for the attestation process are provided, while all unrelated or confidential details remain protected. In other words, the solution significantly reduces the risk of exposing sensitive information in the Measurement Logs by ascertaining whether the entry is related to a pod that an authorised Verifier is allowed to validate. The Master node acts as a proxy, interacting with the Tenant component to register the Agent with the Verifier. It also manages the registration of pods with the Verifier by sending lists of expected pod UUIDs and determining pod status based on the Verifier's evaluation. This approach is particularly beneficial in large, distributed Kubernetes deployments with multiple verifiers, as it prevents unauthorized access to data outside a verifier's scope.

The last IMA template, designated as `ima-cgpath`, was developed to differentiate measurement log entries by associating each entry with its specific pod through its unique UUID, which is embedded in the control group path. This configuration helps the collection of measurements relevant to a specific Verifier by the Master node, identifying pod entries and retrieving their associated UUIDs. To permit this, the template incorporates a field for the control group path, and a dependencies field to understand the related process that generates this entry. However, to further enhance privacy, the introduction of a new template is essential, and all details are described in the section 7.3. This enhanced template incorporates a salt field, enabling the generation of an unpredictable template-hash field, as it prevents unauthorised inference of Measurement Log data. The incorporation of these new Agent capabilities will facilitate the development of refined filtering mechanisms that will enable the isolation and retrieval of logs based solely on the pod ID present. The salted template hash ensures that the filtering process prevents the potential attacker from gaining a measurement traceable to other, previously observed ones.

This solution's design is described through two main processes: *registration process* and *verification process*. The registration process described in the section 7.5 establishes the foundational trust framework by having the Master and Verifier register with a centralized Registrar. This process defines the roles of the components, authorizes verifiers to attest specific pods, and determines the scope of their attestation, ensuring clear boundaries and controlled participation in the verification process. The verification process described in the section 7.6 instead, is initiated by the Verifier to check the integrity of specific pods, corresponding to the remote attestation procedure within the presented scenario. First, the Verifier authenticates with the Master, which checks the verifier's authorization and determines the Worker nodes where the pod is deployed. The Master forwards the request to the Workers, who compute the filtered Measurement List to the Master and provide the Integrity Reports for attestation. The Master aggregates this data into a unified state and delivers it to the verifier, ensuring an attestation result that reflects the integrity of the pod across all involved Worker nodes. The Verifier only has access to information about the specific pod it is authorized to check and does not receive information about other pods, nor about the geographical location of the Worker nodes.

7.3 Measurement Log Analysis

The most recent IMA template (`ima-cgpath`) developed, is capable of facilitating the reporting of the control group path field in addition to the dependencies field within the Measurement Log entries. The focus here is on the first, which was added with the intention of displaying the UUID assigned to the pod managed by Kubernetes, observing the string of the related control group path. This facilitates the binding of a particular ML entry to the relevant pod, thereby ensuring clear distinction, which is advantageous for subsequent filtering on this list. It should be noted that there may be multiple entries with the same pod identifier but with different containers, in the case of containers belonging to the same pod, or even with the same container identifier in the case of different processes running along the same container. As previously stated, the control group path field helps with pod identification, but a Verifier could still see details about pods that aren't their responsibility, which could cause a privacy issue. Moreover, it has been observed that the template-hash value of certain entries can be identical across reboots, enabling a potential attacker to recognise known measures. The template-hash field represents the hash over the remaining fields beyond the template field. If these fields remain almost constant, without something that changes every time before template-hash computation, it is possible to reconstruct the entry identity. For these reasons, the proposal design involves the following IMA modification for each entry:

1. the addition of a new field called *nonce*, a fresh random number generated for each reboot; it is inserted by the Kernel before the template-hash computation making it unpredictable;
2. if the pod identifier in the ML doesn't match the one requested by the verifier, the record will be made up of two fields: the PCR number and the template-hash; this selective inclusion of information prevents the exposure of unnecessary data, safeguarding sensitive details about other pods.

A new template descriptor, called `ima-slt`, was introduced and will be explained in more detail in chapter 8. This template permits the development of a filtering logic for the Measurement List on the Agent's local machine and then responds with the related Integrity Report. This solves the privacy issue that was raised.

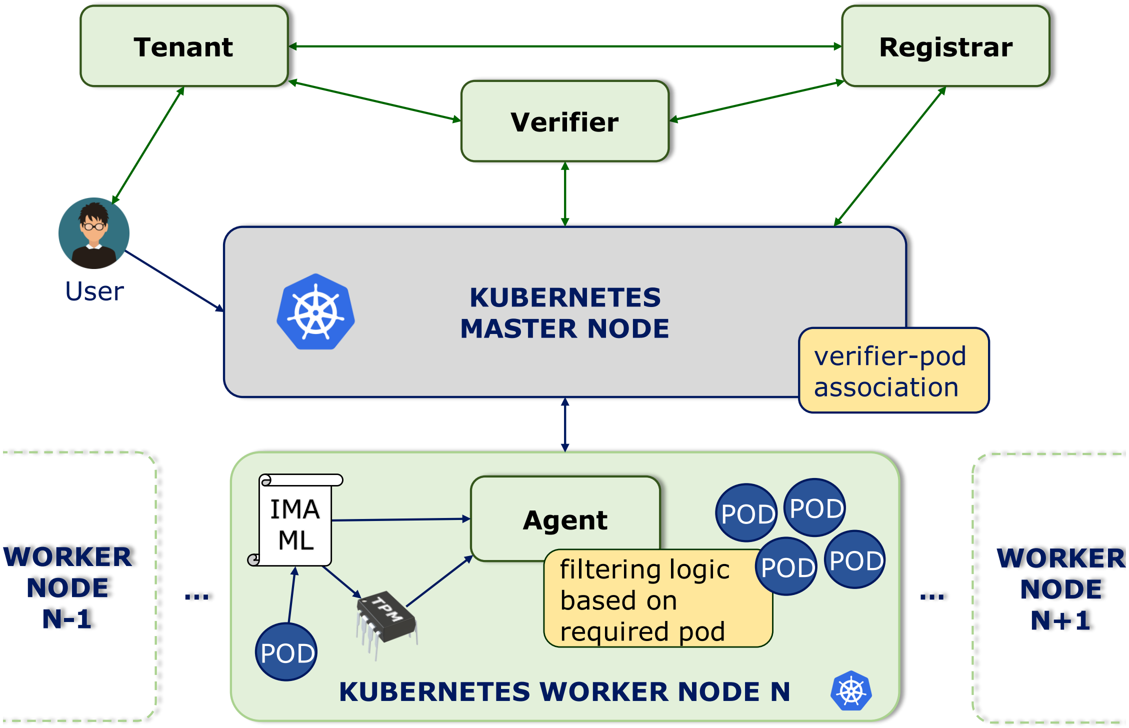


Figure 7.1. Architecture of the proposed solution.

7.4 Architecture of the Proposed Solution

The proposed solution is based on an architectural design in figure 7.1, involving several components, including the tenant, the verifier, the Registrar and the Agent. Each of these components has a specific role to play in ensuring pod integrity in the Kubernetes environment. Outside of the cluster, the tenant is the entity that initiates the attestation process, requesting the verification of specific pods. The verifier, which is also located outside the cluster, is the component that performs the pod integrity check. The Verifier communicates with the Kubernetes Master node, a pivotal component within the architecture that functions as a mediator. In order to operate, it is necessary for both the Verifier and the Master node to be registered with the Registrar, a centralized service external to the cluster that manages authentication keys. The Master node is responsible for the management of authorisations and pod placement and is dislocated inside the cluster with all the worker nodes. The cluster is comprised of worker nodes, each containing an Agent that is responsible for collecting measurements related to pods running on that particular node.

7.5 Registration Process

The registration process interest the main component of the remote attestation, which is intended to facilitate mutual recognition and understanding between participants. This mechanism involves further registration before starting the verification process and revisiting some of the traditional registration procedures Keylime-based needed for remote pod attestation. The following sequence has been established for the ordered registration process together with the figure 7.2:

- It is a prerequisite for the Registrar to enrol before the Master node (1), which is the entity possessing knowledge of the local displacement of their Worker node and registered pods. This part is described in section 7.5.2 with all Master capabilities and other subprocesses necessary for the verification phase.
- Then (2), the process of registering Agents within the Registrar is similar, but there is no direct communication between them. Instead, communication with the Worker node is

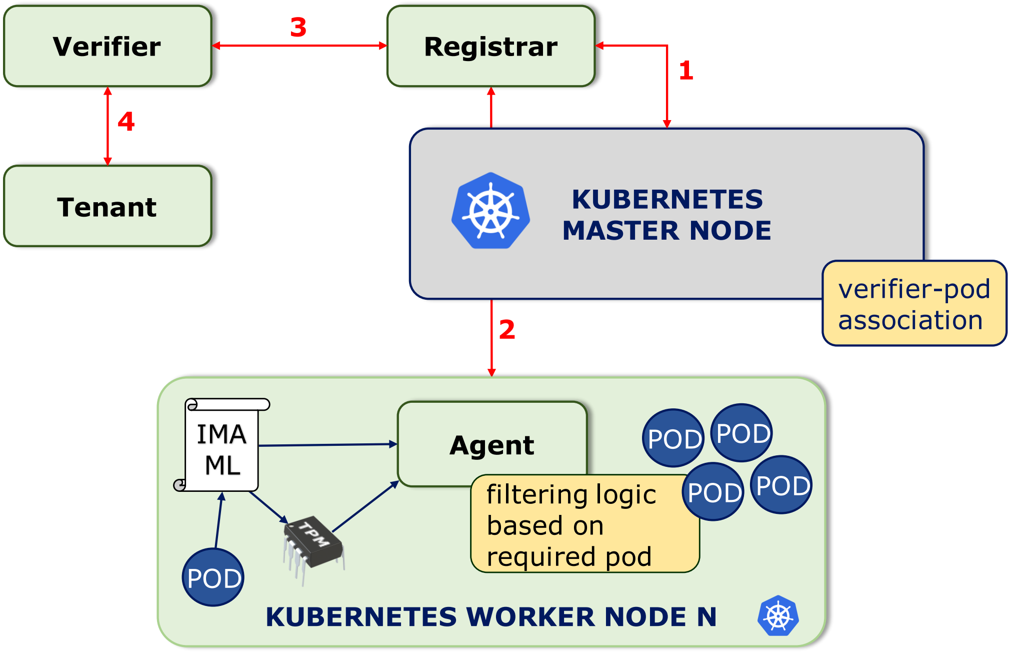


Figure 7.2. Registration order of the components involved in the proposed solution.

passed through the Master node in order to provide its identity. A the end of this point all the involved Agent in each Worker node will be enrolled to the Registrar.

- Once a Master is registered, the focus is on the Verifier (3). In this context, the registration is characterised by the establishment of a unique identifier, designated as the `verifierID` in addition to the IP address, which serves as a reference point for subsequent attestation requests as explained in section 7.5.4.
- Once the Verifier is enrolled, the Tenant can request a Verifier to attest to the integrity of their workload process (4) following the traditional procedure.

In such conditions, all components are known to each other, and a Master can assign the authorisation to an existing Verifier to attest only to the interested pods by following a verifier-pod mapping internally at the Master level. The Tenant can now provide a pod list with all interested `podID` in the Worker node, thus initiating the verification process.

7.5.1 Worker Node

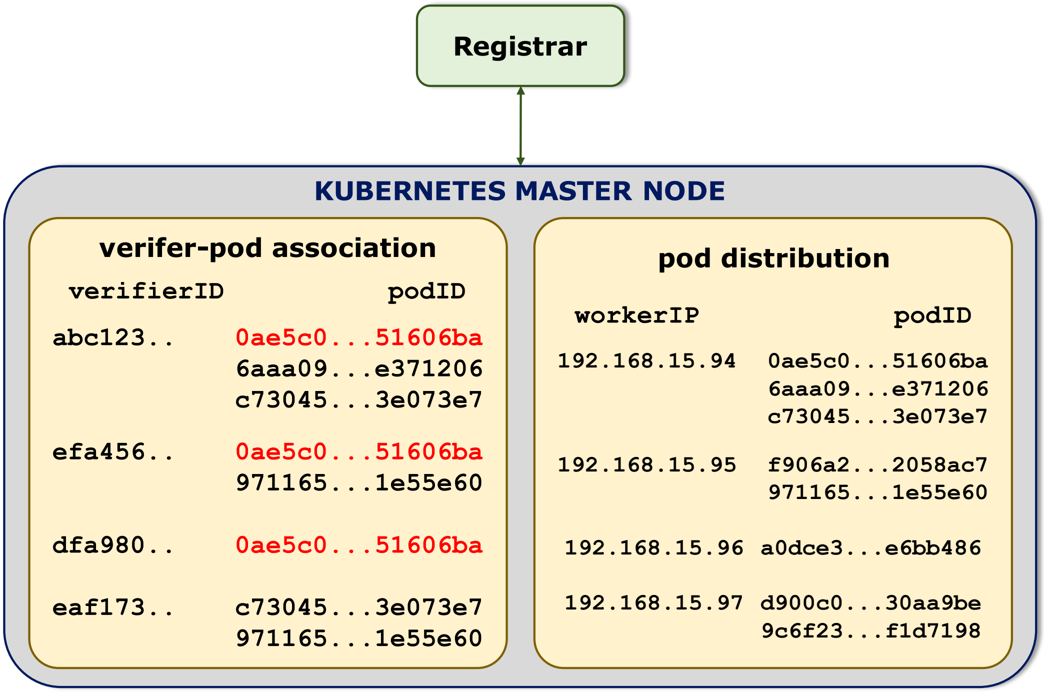
The Worker Node is registered with the Master Node in accordance with the established Kubernetes cluster hierarchy. In particular, in the event of a tenant requesting a new deployment, the scheduler in the Master node will select the Worker node that hosts the necessary pods and will initiate them. Here, the Agent, which runs inside the Worker node just instantiated, must be activated and registered to the Registrar passing through the Master node. Once the Agent has been initiated, it employs the IMA module within the Attester to populate and filter the Measurement Logs to be included in the Integrity Report, utilising the recently proposed `ima-slt` template. This modification is imperative for ensuring enhanced privacy at the Verifier level when the Agent transmits the Integrity Report in response to each attestation request.

7.5.2 Master Node

The Master node plays a central role in the proposed remote attestation framework, acting as a proxy between the Verifier and the Worker nodes. Its registration process begins with enrolling in the Registrar, a centralized service responsible for verifying the authenticity of the hardware platform via TPM's Endorsement Key. The Registrar must be modified in order to host not only keys needed for the attestation mechanism but also information about the Master (such as the IP address) marking it as a known platform. In the event of a remote user requesting a new deployment via the API server, the desired state is recorded in the etcd database. Subsequently, the controller manager is responsible for creating the necessary controllers, while the scheduler determines which physical nodes will host the pods. Once assigned, the Worker node launches the pods using a container runtime. At this stage, the cluster becomes fully operational, continuously working to maintain consistency between the actual state and the desired configuration. The Master node also manages pod registration, a step that ensures that only expected pods are actively running on the Worker nodes. Pod registration involves providing a list of pod UUIDs, retrieved from the orchestrator, that are expected to operate on the node. Each pod is assigned an initial state of START, indicating it is awaiting its first integrity check. As the attestation process progresses, the Verifier updates the pod's state to either TRUSTED or UNTRUSTED, depending on its integrity evaluation.

7.5.3 Master Node Capabilities

Once registered, the Master node takes on a central role in managing authorizations and information about pods. It maintains an internal mapping that links workloads (pods) to verifiers authorised to attest to them. This mapping is of the type `podID:verifierID` and allows the Master to verify whether a given Verifier is authorized to attest to a specific pod. This is a fundamental aspect of the system's security and privacy, as it ensures that only authorized verifiers can access information related to the pod's integrity.



verifer-pod association

verifierID	podID
abc123..	0ae5c0...51606ba 6aaa09...e371206 c73045...3e073e7
efa456..	0ae5c0...51606ba 971165...1e55e60
dfa980..	0ae5c0...51606ba
eaf173..	c73045...3e073e7 971165...1e55e60

pod distribution

workerIP	podID
192.168.15.94	0ae5c0...51606ba 6aaa09...e371206 c73045...3e073e7
192.168.15.95	f906a2...2058ac7 971165...1e55e60
192.168.15.96	a0dce3...e6bb486
192.168.15.97	d900c0...30aa9be 9c6f23...f1d7198

Figure 7.3. Example of pod-verifier mapping and pod distribution.

The Master node also maintains a detailed registry of all Worker nodes within the cluster, providing it with comprehensive oversight of the cluster's structure and resource allocation. This registry includes information about the Worker nodes' identities and their current operational state. Through the use of Kubernetes APIs, the Master can identify the specific Worker node where a particular pod has been deployed procuring information such as pod name, pod IP, the namespace or even the Worker IP address. This list is subject to continuous updating, with the relevant Worker IP address being associated with each pod, since a failure in a pod is addressed with the rescheduling, potentially in another Worker node. Figure 7.3 illustrates the pod-verifier mapping and the pod distribution after the registration phase which can help clarify these relationships. It is important to note that multiple verifiers can be authorised to attest to the same pod.

To preserve system privacy, the Master node operates as a proxy, abstracting internal cluster details from the Verifier. While it holds detailed knowledge of the Worker nodes' local configurations and pod distributions, this information is not directly exposed. The Master node can provide information about pods to an authorised Verifier on demand, but only within strict privacy constraints. Specifically, the Verifier can access details related to the type of workload and processes running within the pod without gaining any insight into the geographical location of Worker nodes or other deployment details. Additionally, the Master would share information about the pod's integrity status, but only after the completed verification process.

7.5.4 Verifier Registration

After the registration of the Master node, the next step is the verifier's registration. In this process, the Verifier is enrolled with the Registrar, thereby establishing its identity and ensuring its recognition within the attestation framework. During the registration process, the Verifier is assigned a unique identifier, denoted as `verifierID`, along with its IP address and all necessary information to be contacted in the network. This identifier serves as a reference point for handling subsequent attestation requests, ensuring that each Verifier is uniquely identifiable within the system. Following the conclusion of the registration process, the Verifier stores the Master node's

IP address as its designated point of contact within the cluster. This enables the Verifier to authenticate with the Master and initiate attestation procedures when requested. It is important to note that the attestation process begins when a tenant requests the Verifier to assess the integrity of specific pods by providing their corresponding `podID`.

The Master can assign authorisation to the verifier, thereby enabling it to attest solely to specific pods. Alternatively, during the deployment of a new workload within a pod, the user responsible for the deployment can specify which Verifier should be authorised to attest to that particular workload. This authorization is provided at the moment of deployment, where the tenant can provide (with the proper framework modification) the Verifier identifier, choosing his favourite considering it the safest. In both cases, the Verifier must be *pre-registered* with the Registrar since it already possesses the necessary information to communicate with the Master.

7.5.5 Verifier Authorisation

Following registration, a Verifier can be authorised to attest to specific pods within the cluster. However, its access is strictly limited to application-level information. The Verifier is only permitted to register a pod in terms of its workload type and application details, without obtaining any information about the underlying infrastructure. This means that the Verifier cannot access information such as the Worker node's IP address, ports, server details or the pod's geographical placement, thereby preserving the privacy of the cluster's architecture. The Master node, in its capacity as an intermediary, retains comprehensive knowledge of the Worker nodes and their deployed pods ensuring that this information remains inaccessible to the Verifier.

The Master node can provide information to an authorised Verifier upon request, but this is subject to strict limitations. The Verifier can access details about the type of workload and application associated with a registered pod, thus ensuring that it can validate the pod's expected behaviour without acquiring unnecessary deployment details. Furthermore, upon completion of the verification process, the Master node is capable of communicating the integrity status of the verifier's assigned pods, thereby confirming their status as either trusted or untrusted.

7.6 Verification Process

Before delving into the step-by-step process, it's important to outline the context. The verification process is how a Verifier checks the integrity of a specific pod and corresponds to the remote attestation process adapted to the presented scenario. It involves several actors: a tenant who requests the attestation, a Verifier authorized to perform the check, the Master node which orchestrates the process and Agents that perform the actual work on the Worker nodes. The process starts outside the cluster, as illustrated in figure 7.4, with the following step:

1. **Attestation Request**: a user initiates the process by sending an Attestation Request to the Tenant, about his application or workload in a user-friendly way.

2. **Forwarding to the Verifier**: the tenant forwards the request to the Verifier to attest one or more of their pods providing a list of UUID to be attested. This step is not included in case two.

3. **Forwarding to the Master**: the Verifier then forwards the tenant's request to the Master node, on the basis that the user is known to the system. The location of the Master is known to the Verifier based on prior registration with the Registrar, and the verifier may or may not be authorised to attest specific pods.

4. **Verifier Check**: the Master node checks if the Verifier is already registered. If registered, the Master node does not forward the request directly to the Agent. Instead, it proceeds to check internal permissions, using the previously established `podID:verifierID` mapping, to see which pods the Verifier is authorized to attest.

5. **Worker Identification**: the Master checks which are the relevant Workers by consulting the pod distribution list continuously updated. If those UUIDs involve multiple entries in the list, different Workers need to be contacted.

6. **Forwarding to the Worker**: The Master forwards the attestation request to the relevant Worker nodes, for instance, the n-Worker.

As demonstrated in figure 7.5, the process continues within the cluster, with the subsequent step being:

6. **Agent Receives Request**: the Agent is responsible for receiving the attestation request on a designated port from the Master node, about a pod or a set.

7. **ML Consulting and Filtering**: the Worker node verifies the trustworthiness of the pods using the same logic as the host machine. Once the integrity of the Measurement List has been confirmed against PCR 10, both the host system and the pods will be validated simultaneously. The TPM quote is determined by each Worker, who calculates the ML associated with the underlying machine and filters it in accordance with specific criteria. The process of filtering is initiated as follows: each entry in the Measurement List is evaluated based on its `podID`. In the absence of a `podID` in the entry, the entry is regarded as relating to the host system and is left intact. The same approach is adopted if the `podID` entry identifier matches the one requested for attestation. Otherwise, only the first two fields are returned, specifically the PCR number and the template hash. The template hash field is mandatory, as this is required to permit the assessment of the system's integrity state at the verifier level.

8. **Filtered ML to Master**: the Agent then transmits the filtered list back to the Master node.

The concluding stages involve a values comparison in order to validate the system's trustworthiness again outside the cluster. Figure 7.6 illustrates these steps in more detail:

8. **Master Receives Filtered ML**: the Master receives the filtered ML.

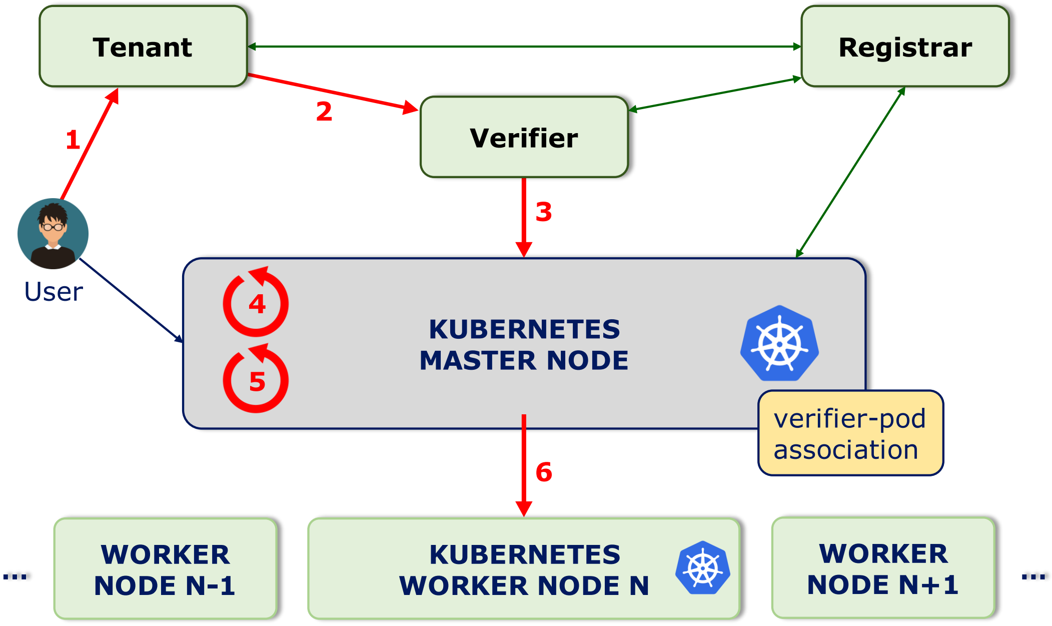


Figure 7.4. Verification process in the proposed scenario part 1.

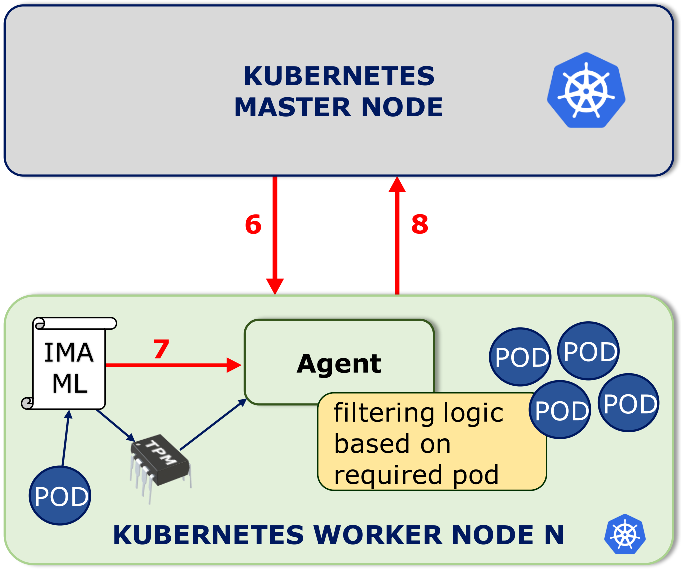


Figure 7.5. Verification process in the proposed scenario part 2.

9. **State Aggregation**: (optional) the Master has the option to aggregate these responses if they are from different Workers.

10. **ML to the Verifier**: the Master forwards the filtered (or aggregated) ML to the Verifier.

11. **ML validation**: the Verifier receives the list and performs validation according to the Remote Attestation procedure.

12. **Attestation Result**: the Verifier generates an attestation result.

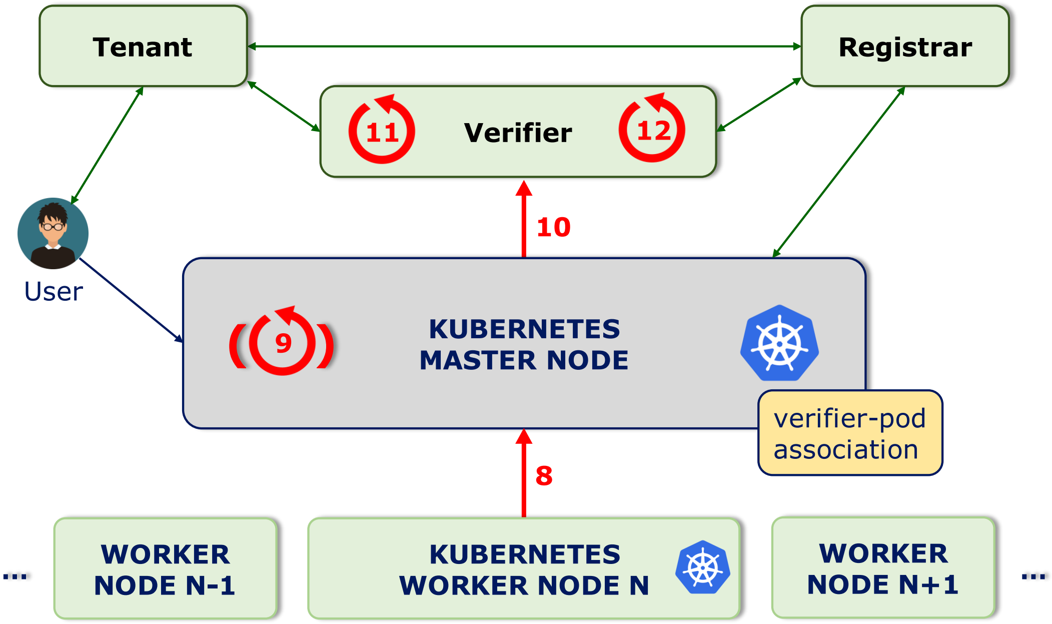


Figure 7.6. Verification process in the proposed scenario part 3.

This process can be performed periodically, as the user must first interact with the orchestrator to obtain the current pod identifiers, which are dynamic and may change over time. In the event of a Worker node failure or shutdown, the affected pods will be rescheduled with new UUIDs and may be deployed on one or more different Worker nodes. It is evident that the operation in question is executed by the scheduler. Consequently, the master is invariably informed of the pod distribution, as the scheduler is responsible for the continuous monitoring of this list. Following the retrieval of the updated pod UUIDs, the user then contacts the Tenant to initiate a new Attestation Request, thus restarting the verification process.

7.7 Privacy and guessing analysis

The result of the verification process is that the Verifier can verify the integrity of a pod and the Worker it runs on without knowing information about other pods. This information remains private to the cluster administration. During the verification process, the Verifier receives a filtered ML containing information related to the pod for which it requested attestation. This list is generated by the Agent within the Worker node and subsequently processed by the Master node. It is important to observe that the Verifier does not obtain direct information about other pods running on the same Worker node or other Workers. The only data that the Verifier receives, besides the integrity of the verified pod, is the first field of the list (template-hash), which is always different due to the addition of a salt, making it impossible to relate this information to known measurements.

A potential attacker, by analysing the filtered and obfuscated entries of the ML, might try to guess the amount of work performed by the Worker on which the requested pod is running. The attacker might also infer that other pods are running on the same Worker without being able to determine the exact number of such pods. The relative proportions of work performed by a single pod or multiple pods remain uncertain. A single pod is capable of performing a significant amount of work, whereas multiple pods are capable of performing a smaller amount of work.

The attacker cannot obtain specific information related to other pods, such as the type of application, dependencies, or the container runtime. Information regarding the geographical location of the Worker is also hidden from the attacker who does not know if and where the pod has been located. Information about the location and configuration of the pods remains private to the cluster administration. Instead on the tenant side, details on pod placement and configuration are protected and not accessible to the user consuming the service. This ensures that information related to the cluster's internal workings remains confidential to the administration, maintaining the privacy of the system.

Chapter 8

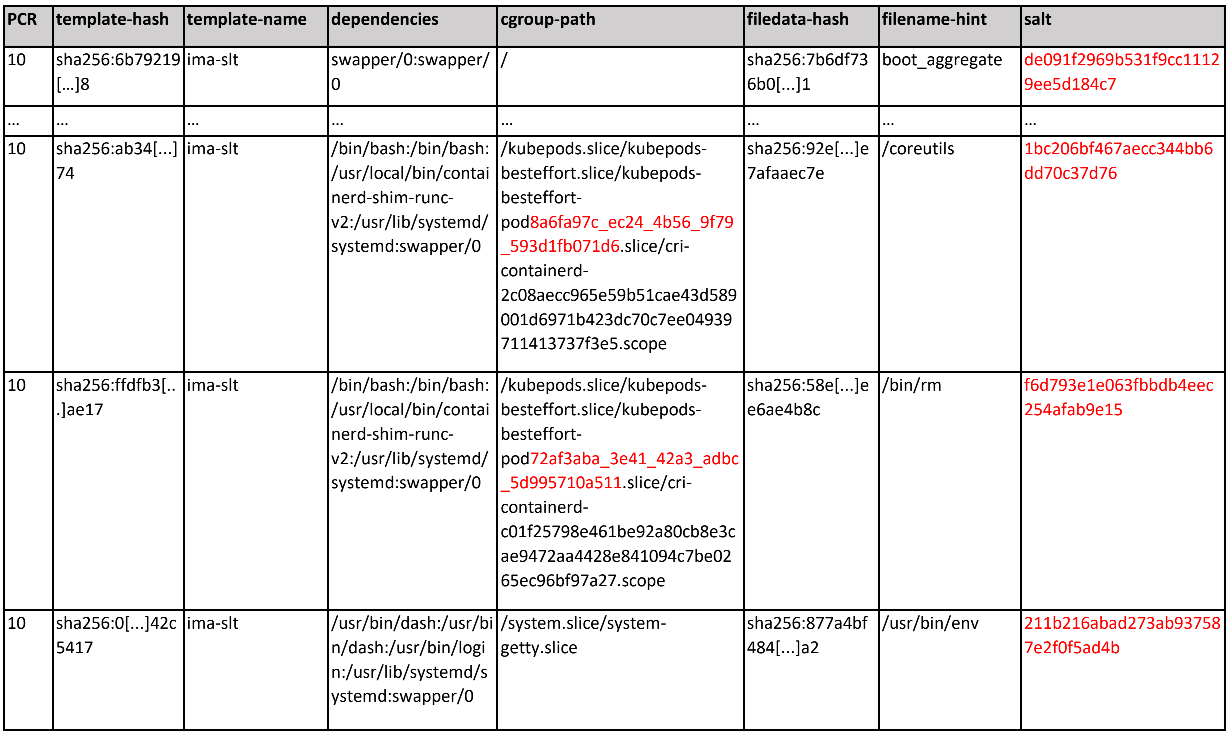
Proposed solution - Implementation

This chapter details the implementation of the proposed solution, with a particular focus on enhancing remote attestation mechanisms within Kubernetes. The solution is built upon modifications to both the Linux Integrity Measurement Architecture and the Keylime framework, along with additional Kubernetes capabilities to improve security and privacy. The chapter begins by discussing modifications to the IMA template and kernel, which ensure that Measurement Logs accurately represent pod-specific information. It then details the enhancements introduced to Keylime, a critical component in the attestation process. These modifications encompass several Keylime components, including the Tenant, Agent, Registrar, and Verifier, each of which contributes to the enhancement of integrity verification enforcing privacy outside the cluster. The remaining sections of the chapter proceed to examine Kubernetes integrations for secure and efficient attestation. These include a proxy implementation for centralized component interaction, an authorisation mechanism to grant or reject data access, and a feature to build a compressive state of the ML from different attester machines.

8.1 IMA Template and Kernel modification

The recently refined IMA template is predicated on its predecessor, with particular emphasis placed on the entry discrimination corresponding to the host system and those associated with a pod. This was made possible by the `cgroup-path` field, which displays the associated UUID if the entry is related to a pod. The recently suggested template incorporates an additional field corresponding to a nonce for each entry. The work operates under the assumption that a fresh 32-bit random number is adequate for the purpose of generating unpredictability. The template descriptor is referred to as "`ima-slt`", accompanied by a format string that reads "`dep|cg-path|d-ng|n-ng|slt`". The abbreviation "slt" is employed to denote the additional field, which functions as a *salt* for the template-hash computation and hence is subject to change following a system reboot. Referring to Figure 8.1, the template fields utilised for each record are as follows:

- *PCR* as the index of the PCR at which the entry was extended;
- *template-hash*, the representation of the digest extends from the specified PCR, calculated over the template fields, from the template name to the last field; the SHA-256 algorithm is utilised;
- *template-name* specifies the template descriptor used for the entry. In this case, it is `ima-slt`;
- *dependencies* of the process that resulted in the creation of this event;



PCR	template-hash	template-name	dependencies	cgroup-path	filedata-hash	filename-hint	salt
10	sha256:6b79219[…]8	ima-slt	swapper/0:swapper/0	/	sha256:7b6df736b0[…]1	boot_aggregate	de091f2969b531f9cc11129ee5d184c7
…	…	…	…	…	…	…	…
10	sha256:ab34[…]74	ima-slt	/bin/bash:/bin/bash:/usr/local/bin/containerd-shim-runc-v2:/usr/lib/systemd/systemd:swapper/0	/kubepods.slice/kubepods-besteffort.slice/kubepods-besteffort-pod8a6fa97c_ec24_4b56_9f79_593d1fb071d6.slice/cri-containerd-2c08aecc965e59b51cae43d589001d6971b423dc70c7ee04939711413737f3e5.scope	sha256:92e[…]e7afaaec7e	/coreutils	1bc206bf467aecc344bb6dd70c37d76
10	sha256:ffdfb3[...]ae17	ima-slt	/bin/bash:/bin/bash:/usr/local/bin/containerd-shim-runc-v2:/usr/lib/systemd/systemd:swapper/0	/kubepods.slice/kubepods-besteffort.slice/kubepods-besteffort-pod72af3aba_3e41_42a3_adbc_5d995710a511.slice/cri-containerd-c01f25798e461be92a80cb8e3cae9472aa4428e841094c7be0265ec96bf97a27.scope	sha256:58e[…]ee6ae4b8c	/bin/rm	f6d793e1e063fbbdb4eec254afab9e15
10	sha256:0[…]42c5417	ima-slt	/usr/bin/dash:/usr/bin/dash:/usr/bin/login:/usr/lib/systemd/systemd:swapper/0	/system.slice/system-getty.slice	sha256:877a4bf484[…]a2	/usr/bin/env	211b216abad273ab937587e2f0f5ad4b

Figure 8.1. Example of ML with ima-slt template.

- *cgroup-path* contains the pathname of the cgroup about a particular container within a specific pod;
- *filedata-hash* contains the digest of the specified file; regards as the actual measure of the event obtained utilising the SHA-256 algorithm;
- *filename-hint* is the conventional file path name;
- *salt* is defined as a 32-bit random number generated each time the system is restarted.

All fields previously present in the preceding template are mandatory, while salt has been newly introduced. It is important to note that the template-hash field utilises the SHA-256 algorithm, whereas the default hashing algorithm employed in the IMA module is SHA-1. Originally introduced by Docker Container Attestation developers, this enhancement has been adopted in the current solution, allowing the hashing algorithm to be reverted to SHA-1 via the `ima_template_hash=` kernel parameter. The patch implementation of the new template is set out in section B.1 of the Developer's Manual.

At the Agent level, this table is fully available. By observing the `cgroup-path` field, it is possible to identify the pod identifier and the specific container in which the process is executing. Utilising these details, a horizontal filtering logic has been implemented, retrieving the podID from the Verifier request and selecting a specific subset of the entry. The term 'horizontal' signifies that only the first two fields are returned to the Verifier if the entry is not of interest. The motivation behind the introduction of the final field is to induce unpredictability due to the inherent variability of the salted template-hash value, which renders it indistinguishable from conventional measures.

8.2 Keylime modification

The Keylime modifications pertained to two primary considerations. Firstly, the objective was to ensure the framework was both capable of understanding and utilising the newly developed IMA template. This was to be implemented at the Agent level, responsible for the management of the Measurement Log, as well as at the Verifier level, which is responsible for the acceptance of a list comprising entries that potentially contain fewer fields than expected. Secondly, given the function of the Kubernetes Master node as an intermediary between the Agent and other framework components, the modification must ensure that all requests and responses in the attestation process are correctly routed and managed following the established consensus.

Upon instantiation of the IMA template in the attester machine, the Keylime framework initiates the registration process, which involves the components in the order proposed in the design section. Initially, the Master node must be registered to the Keylime Registrar, providing the necessary information to be contacted over the network, as outlined in section 8.2.3. Subsequently, the process of registering Keylime Agents within the Registrar is analogous to the Physical Node Registration Protocol employed by Keylime, with the distinction that a Master mode now exists between them and that communication must be routed to the appropriate destination. This necessitates modifying the port and IP address, which Keylime allows easily by editing specific fields in the configuration files. A similar process is required when Keylime Agents register with the Verifier component as part of the Three Party Bootstrap Key Derivation Protocol, a process that also involves the Tenant informing the UUID of the Agent under examination. Given that the Registrar is situated outside the cluster, an “Ingress” Resource is required to enable communication with the Keylime Agent together with other services offered by Kubernetes outlined in greater detail in section 8.3. Generally, some modifications are required for each component, which will be described individually in the following sections.

The architecture of the component in question, as it pertains to the implementation, is delineated in Figure 8.2.

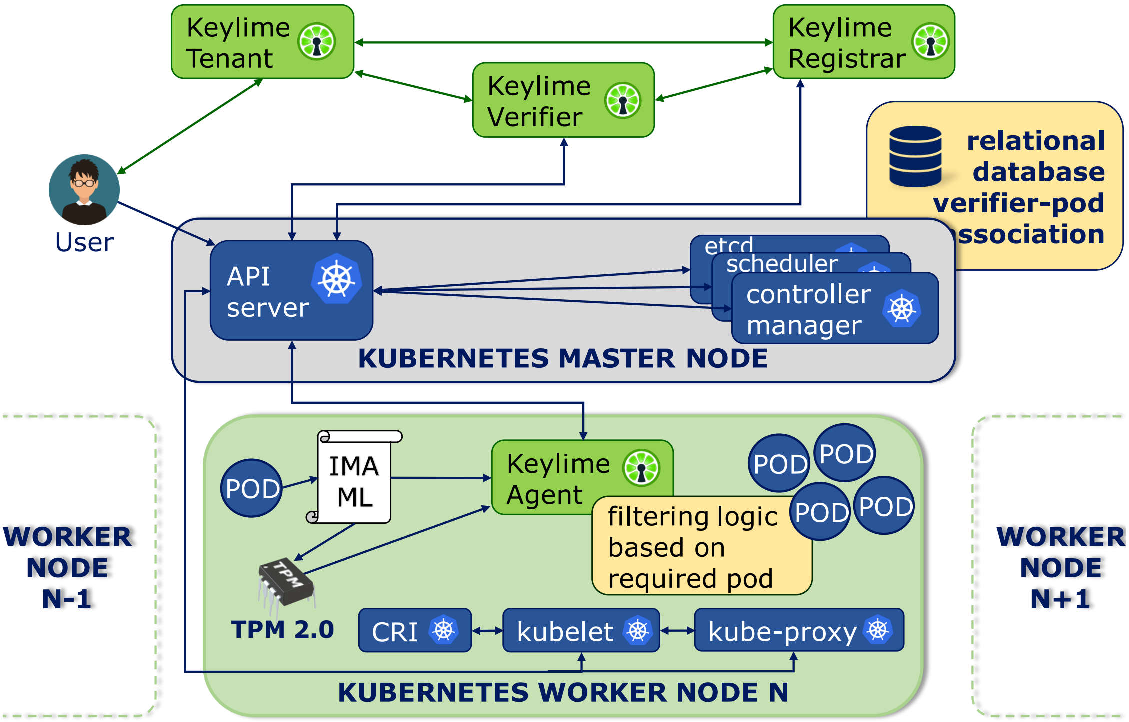


Figure 8.2. Overview of the proposed architecture.

8.2.1 Tenant modification

Tenants represent a tool registered to the framework and possess a list of UUIDs corresponding to the pods to be attested. The Agent provides a set of APIs for tenants that offer several critical functionalities. Firstly, these APIs allow tenants to monitor the operational status of the Agent as well as the pods running on a given Worker node, ensuring continuous visibility of the current state. Additionally, tenants can dynamically add, modify, or remove pods from the attestation list, providing flexibility in managing pod registration. Alternatively, a Tenant can query the integrity status of an individual pod with proper APIs for instance by issuing a GET request with the pod's UUID; the Verifier will then respond with a status of either START, TRUSTED, or UNTRUSTED. These features, as previously introduced in earlier works, are fundamental for this thesis work and require no further modifications. The only modification to enforce is to change the IP address of the Agent to the Master IP address in order to ensure the transmission of the correct information to the Verifier, which will then route the Attestation Request correctly. It should be noted that the Agent's UUID to be transmitted remains invariable and will be recognised at the Master level.

8.2.2 Agent modification

The Keylime Agent periodically transmits an Integrity Report, encompassing the Measurement Log, to the Keylime Verifier. To provide support for the IMA template that has been recently introduced, modifications were made to the Kernel code responsible for handling IMA interactions. These modifications included the addition of a new class, named `ima-slt`, in the attester machine, which permits the populating of the ML with the additional field that has been proposed. A further significant modification pertains to the filtration of the Measurement Logs by the Keylime Agent. The Agent is responsible for exposing an API as the point of contact to receive requests, along with the UUID of the pod to attest. These APIs have been developed in the context of previous thesis works and thus do not necessitate modification, given their basis on the pod identifier. The possession of the UUID facilitates the enforcement of the filtering logic illustrated in figure 8.3. Before computing the KeylimeQuote, the process commences with a comprehensive scan of all ML

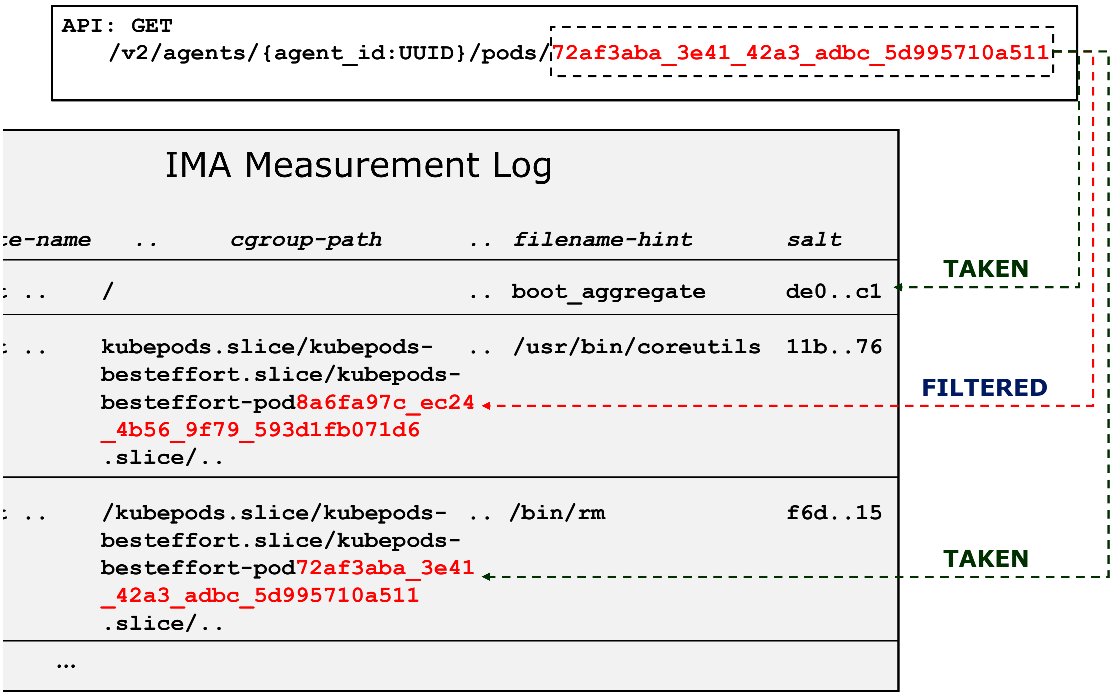
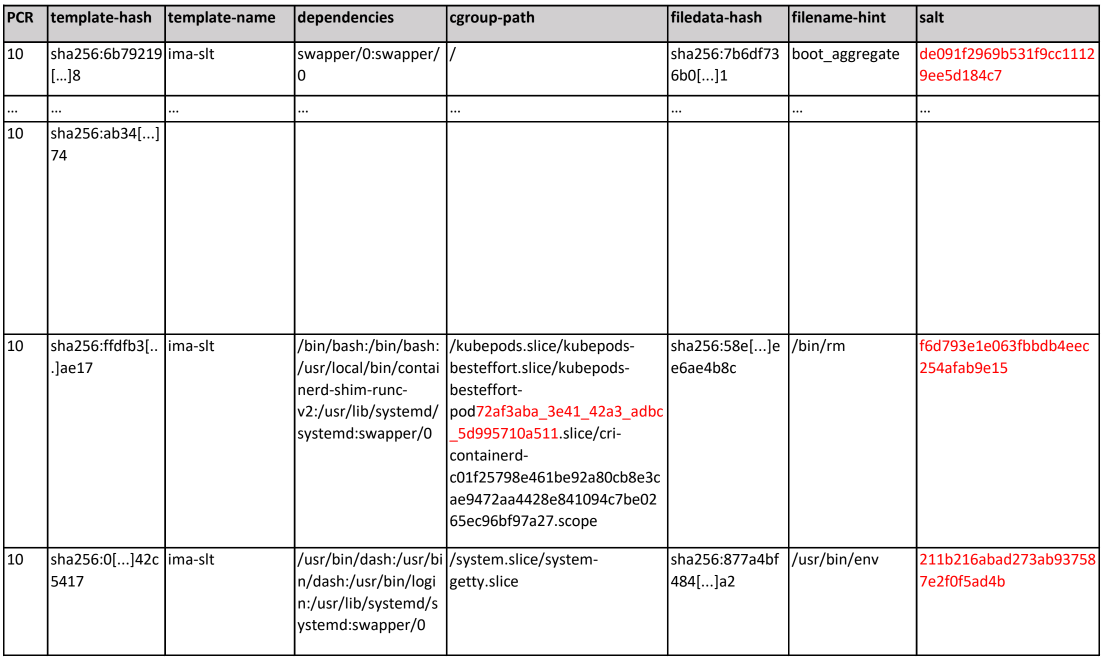

API: GET
/v2/agents/{agent_id:UUID}/pods/72af3aba_3e41_42a3_adbc_5d995710a511
IMA Measurement Log
:e-name
..
cgroup-path
..
filename-hint
salt
: .. / .. boot_aggregate de0..c1
: .. kubepods.slice/kubepods-besteffort.slice/kubepods-besteffort-pod8a6fa97c_ec24_4b56_9f79_593d1fb071d6.slice/.. .. /usr/bin/coreutils 11b..76
: .. /kubepods.slice/kubepods-besteffort.slice/kubepods-besteffort-pod72af3aba_3e41_42a3_adbc_5d995710a511.slice/.. .. /bin/rm f6d..15
...
TAKEN
FILTERED
TAKEN

Figure 8.3. Filtering logic of the Keylime Agent.



PCR	template-hash	template-name	dependencies	cgroup-path	filedata-hash	filename-hint	salt
10	sha256:6b79219[…]8	ima-slt	swapper/0:swapper/0	/	sha256:7b6df736b0[…]1	boot_aggregate	de091f2969b531f9cc11129ee5d184c7
…	…	…	…	…	…	…	…
10	sha256:ab34[…]74						
10	sha256:ffdfb3[...]ae17	ima-slt	/bin/bash:/bin/bash:/usr/local/bin/containerd-shim-runc-v2:/usr/lib/systemd/systemd:swapper/0	/kubepods.slice/kubepods-besteffort.slice/kubepods-besteffort-pod72af3aba_3e41_42a3_adbc_5d995710a511.slice/cri-containerd-c01f25798e461be92a80cb8e3cae9472aa4428e841094c7be0265ec96bf97a27.scope	sha256:58e[…]ee6ae4b8c	/bin/rm	f6d793e1e063fbbdb4eec254afab9e15
10	sha256:0[…]42c5417	ima-slt	/usr/bin/dash:/usr/bin/dash:/usr/bin/login:/usr/lib/systemd/systemd:swapper/0	/system.slice/system-getty.slice	sha256:877a4bf484[…]a2	/usr/bin/env	211b216abad273ab937587e2f0f5ad4b

Figure 8.4. Example of filtered ML with ima-slt template.

entries and checks if each `cgroup_path` value contains a specific pattern to respect. The pattern is defined as `"/kubepods.slice/kubepods-besteffort.slice/kubepods-besteffort-pod"`, `pod_UUID`, `".slice"` and if the `pod_UUID` matches the received one in the API, the entry is taken entirely. On the contrary, verification is conducted to ascertain whether the value of `cgroup_path` encompasses the initial segment of the pattern. In the event that the initial segment is present, it can be deduced that the entry is associated with an event originating from another pod. Consequently, the initial two entry fields, the `template-hash` and the `PCR` number, are selected. Otherwise, the entry is tied to an event generated by the underlying host system, and again, the entry is taken entirely since the Verifier has to know details about the machine to attest. The resulting table is illustrated in figure 8.4 and will be communicated firstly to the Master and subsequently to the Verifier. It is worth noting that all entries pertain to a particular Worker node on a specific IP address; instead, all entries received at the Master level are derived from other machines within the cluster, and therefore, an aggregation state is necessary. The described script is available in the section B.2 of the Developer's Manual.

8.2.3 Registrar modification

First, it is necessary to modify the Keylime Registrar to enroll the Kubernetes Master node network details. Specifically, the IP address and port of the network node are to be saved to inform the Verifier about the right destination when it has to be enrolled later. These data are stored in the same database that is used for the attestation process, and in the field designated for the IP address of the Agent, rather than the one for the Master. This operation is performed before the Physical Node Registration Protocol between the Agent and the Registrar, and this field is not altered in subsequent operations. In summary, the information regarding the Agent destination is not retained and is instead substituted by the Master destination. The Master will subsequently forward the information to the designated Agent recognising the associated UUID.

8.2.4 Verifier modification

The Keylime Verifier must be modified to support the new version of the Measurement List provided in the Integrity Report, without the risk of erroneous results. The Verifier is in charge of the IMA ML validation process, which is performed by checking each entry in the list. It parses each entry by splitting it into fields using whitespace as a delimiter since the Agent transmits the IMA Measurement Log in ASCII. By default, Keylime utilises the template name from the third field to determine which template to apply and computes a digest of the corresponding template fields using the hash algorithm specified in the Integrity Report. Although the template-hash is already the digest computed over these fields beyond the template-name, for best practice, Keylime checks if these values are the same. If these values are found to be different, the Verifier will log the discrepancy identified during the ML verification process. However, it should be noted that with the new version of the list, these fields in entries that are not interesting are missing. For this reason, modifications are required to shift the validation to bypass the recalculation of the digest and to rely on the template-hash value present in the entry. Consequently, the Verifier performs an "extend" operation by combining the template-hash with the aggregated PCR, known as the `running_hash`, which is built incrementally as entries are processed. Upon completion of the processing of the entire list, the Verifier then compares the `running_hash` to the IMA PCR value provided in the quote. A match between these values serves to confirm that the ML has not been tampered with up to that point. To m For the primary attestation, the `running_hash` is initialised to an all-zero string since the entire Measurement Log is received. For subsequent attestations, it resumes from the IMA PCR of the previous attestation because only the new ML entries are transmitted. After verifying the Measurement Log's overall integrity, the Keylime Verifier validates each entry by consulting the appropriate host and pod whitelists. In the event of the validation of measurement log integrity encountering failure, or in the event that one or more measures do not correspond with those present in the whitelist, the Attester system is evaluated as untrusted. The snippet of code of interest for the new class and validation is shown in section B.3 of the Developer's Manual.

8.3 Kubernetes Capabilities integration

This section provides an in-depth description of the implementation, encompassing all features at the Master level, with the use of different scripts.

8.3.1 Proxy Feature

In the proposed solution, the Master should act as a proxy, thus preventing other communications from contacting nodes in the internal network. Specifically, the solution necessitates the implementation of a reverse proxy that incorporates designated filtering policies. The reverse proxy is an intermediary component between the communication infrastructure and the available services. It receives incoming requests from clients on the external cloud and subsequently forwards them to the available services. In this particular instance, the service in question is the Keylime Agent responsible for providing the quote associated with the required pod to attest. This feature can be enforced by a Kubernetes resource, named *Ingress*, which is a service available through the utilisation of a protocol-aware configuration mechanism. This mechanism possesses the capacity to comprehend web concepts such as URIs, hostnames, or paths, and within the Ingress resource, facilitates the mapping of traffic to distinct back-ends, based on rules defined via the Kubernetes API. As illustrated in Figure 8.5, the present resource and the other associated ones are situated within the proposed architecture. In order to utilise this service, it is necessary to define a particular YAML file (of the Ingress `kind`) which specifies the domain of the Master node's public address with the listening port. It is also necessary to specify the API exposed by the Agent in the `path` field, together with a snippet of code on which a forward logic and the authorisation logic work with the related file name. A service of the type *NodePort* must be defined here. It constitutes an additional resource that has been defined previously, encompassing not only the specifications of the incoming communication but also the `nodePort` number on which the Keylime Agent on the Worker node is waiting to receive requests. For each Agent, a `ClusterIP` Resource is enforced to process the incoming TCP traffic from the conventional port and choose the right `targetPort` as pod destination. It is imperative to note that this Service is defined individually for each Worker node, with each node possessing a distinct namespace. The files in question are interconnected within the system through a label match. At this stage, the emphasis is on the logic that determines whether to forward the integrity request to the appropriate Worker node. For the present scope, a Python script has been developed that discriminates the pod UUID received from the Verifier or a list of pods and forwards the received request to the right Worker node. The Python script performs a series of actions, the full details of which are outlined in section B.4 of the Developer's Manual. Firstly, it identifies the location of the pod by consulting the pod distribution list, a process that is continuously updated by Kubernetes. Secondly, it builds a request with all the required details. To execute this script, it is necessary to launch a pod using a `Deployment` resource, which runs it. It includes a defined `containerPort`, through which the service is accessible within the cluster, utilising a `ClusterIP` resource also in the Master node. This deployment is defined in a YAML file, which specifies the deployment, with a base `image: python:3.8-slim`, on which a container is able to execute a Python script. In the end, the script is included in the image thanks to a custom `Dockerfile`, which contains it and all the necessary packages. A comprehensive YAML definition may be found in the section A.2.3 of the User's Manual.

8.3.2 Verifier Authorisation

The Verifier authorisations refer to the mapping `verifierID:podID` presented in the design chapter. This mapping is implemented in a `sqlite3` database local to the Master on which all the associations are available. Associations are recorded as records in a table containing just two fields: the `verifierID` and the `podID`. Given the possibility of different verifiers attesting the same pod, or of different pods being attested by the same Verifier, the relation is N:M, and the primary key of the table encompasses both fields. The authorization logic is implemented in the same file that is used for the forwarding logic, and it is enforced before the request is sent to the correct Worker node. In order to provide a more detailed description of the process, it is first necessary to explain the function of the initial check. This check ensures that a given Verifier is already present in the system by querying the database using a SELECT statement to ascertain

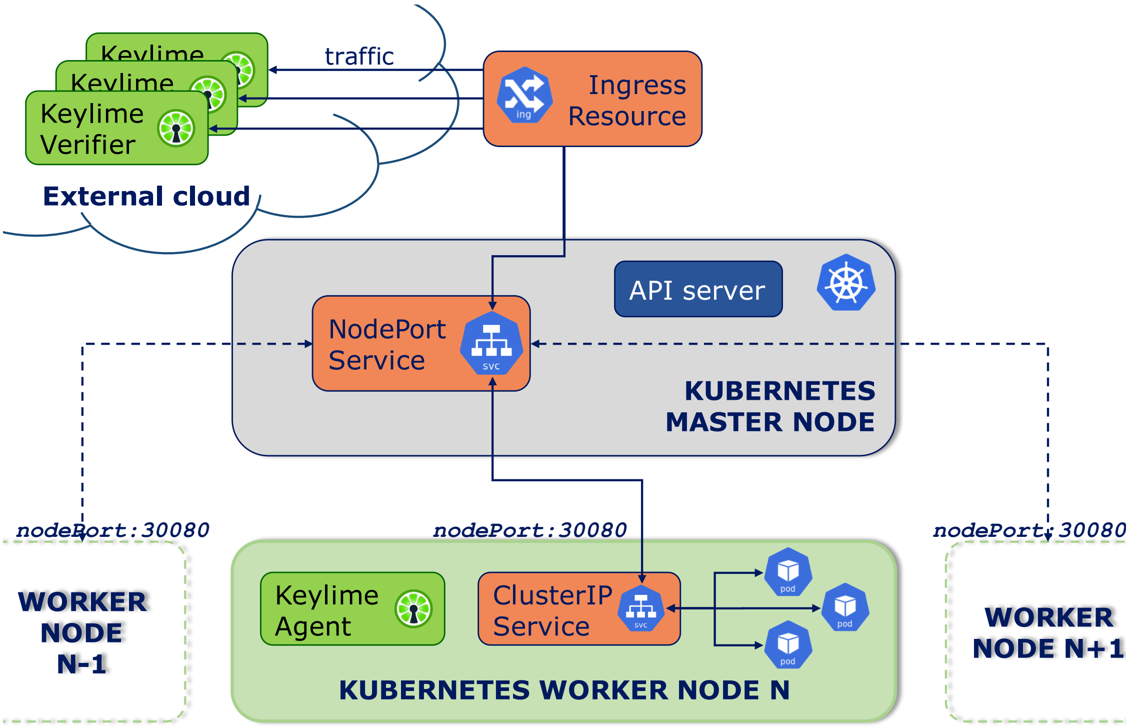


Figure 8.5. Resources utilisation in the proposed architecture.

the identifier's existence. In the event that the identifier is present, the script continues; otherwise, the attestation attempt is halted immediately. Following this, authorisation to contact the Worker is granted after a successful check on the mapping over the `verifierID` and `podID` provided. In essence, when the pod list provided is scanned in the previous Python script, this verification is performed for each entry by consulting the database to ascertain whether there is a record with both identifiers. This is achieved through a query with a SELECT statement, as previously described. In the absence of the record, it can be deduced that the Verifier lacks the requisite authorisation to attest to this particular pod. Consequently, the system issues an error message, thereby impeding the attestation attempt. On the other hand, the request is correctly forwarded to the right Worker node following the specifications presented in the previous section.

For the simplicity of the process, the database is initially empty and will be populated each time a workload to be attested is deployed. In the proposed solution, a new Python script is developed to substitute the command to apply a workload configuration used by Kubernetes through a YAML file. The new script is implemented to function similarly to the standard, such as when YAML files are used in the `kubectl apply -f` command, with the additional step of specifying the `verifierID` after the `-verifier` tag. The code first initiates deployment in the customary manner employed by Kubernetes, subsequently retrieving the list of pod UUIDs associated with the scheduled pods. Following the compilation of said list, the pods are registered using an INSERT query to the database utilised for the purpose. This query is repeated for each pod, with the number of iterations corresponding to the number of generated pods. The pod UUID and the Verifier identifier at the conclusion of this process will be registered in the database in each record. A segment of code for the designated apply command is presented in section B.5 of the Developer's Manual

8.3.3 State Aggregation Feature

Once the `KeylimeQuote` has been built with the filtered Measurement List, each Agent responds to the Attestation request with a response related to the required pods. Each response is transmitted to the designated port on a specified IP address, as outlined in the configuration file. In this particular instance, the port is monitored by the Master node within the same network, and it is listening for the response by the Keylime Agent. In particular, the Master node is waiting to receive many responses as many requests are already forwarded to Worker nodes. When the Master collects all the received responses, it takes care of the serialization of an object composed of a set of quotes received. This response is forwarded to the Verifier in the usual method since it exposes the API to retrieve the quote from the Agent. The result of the attestation will be determined based on the state aggregation provided. The present implementation is based on a prior script when the forwarding logic is initiated, where the Master invokes itself as an API, transmitting the `verifierID` and the anticipated quantity of pods to be attested. All the APIs exposed by the Master have been constructed using `Flask` (a lightweight web framework for Python), which has the purpose of facilitating the process of building APIs. Accordingly, the system has been configured to await many responses corresponding to the number of Verifier identifiers requested. The implementation of this waiting is achieved through the use of a `threading` Python library utilised to facilitate synchronisation between the function that receives the response from the Agent and the function that calculates the state aggregation. The first function is invoked each time an Agent transmits a quote to the Master node, appending the response to an array of responses. The second one is waiting for a condition on the `expected_responses` variable, defined when the Attestation Request is sent to the Agents. The condition consists of waiting until the number of expected responses is equal to the length of the array of the received response. Once this condition is met, the primary process acquires all the responses collected at that particular instant and transmits them to the Verifier. The Verifier listens to the response via a TCP service script on a specific port, without exposing any API.

Chapter 9

Tests

The present chapter concerns the testing and validation phase of the proposed architecture. In this regard, a series of tests has been devised to ensure the system's correctness, reliability, and performance under various conditions. In particular, functional tests are conducted to verify the correct behaviour of the remote attestation process, particularly focusing on scenarios involving legitimate and unrecognised Verifiers and pods. Concurrently, the evaluation encompasses a comparison with the standard Keylime framework to assess the architecture's capacity to perform remote attestation correctly in the proposed environment. In the end, performance tests are utilised to analyse attestation times, thereby measuring the system's efficiency and scalability in relation to the growth in the number of pods.

9.1 Testbed

The prepared testbed for the proposed solution is composed of two machines:

- A *Kubernetes Master Node* where the Keylime Registrar, the Keylime Verifier and the Keylime Tenant are launched. The machine is an Intel NUC with an Intel Core i5-5300u CPU, and the operating system used is Ubuntu Server 24.04.2 LTS.
- A *Kubernetes Worker Node* as the attester machine running the Keylime agent. The machine is an HP Elitebook laptop equipped with an Intel Core i7-7600u CPU and a real HW TPM module manufactured by Infineon. Ubuntu Desktop 22.04.1 LTS is the OS used, with a Linux kernel patch applied to a specific branch checkout using the new IMA template explained in section 8.1.

9.2 Functional Tests

The first step in the testing process is to understand the correctness of the proposed solution within the architecture through functional testing. If the host system and each pod running on it match their respective whitelist values, the Keylime Verifier is expected to consider both the host system and each pod as trusted. The test considers running the pod with the script to externally expose the API for receiving and forwarding requests, and already granting permission to a verifier to attest specific groups of the pod. On the Kubernetes worker node, the Keylime agent was started using the default UUID `d432fbb3-d2f1-4a97-9ef7-75bd81c00000` and a single nginx pod scheduled by the master, with the following `nginx-deployment.yaml` file:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

All the test performed start with the Verifier running this command on the terminal with its unique `verifier_id` to attest a specific pod indicated as a `pod_id` in the body of the JSON request:

```
curl -X POST http://localhost:5000/verify -H "Content-Type: application/json" -d '{
    "verifier_id": "default",
    "pods": [
        {
            "pod_id": "aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa",
            "whitelist": [],
            "exclude_list": []
        }
    ]
}'
```

For simplicity, Keylime Verifier runs on the same machine as the Kubernetes master node, and the API is exposed internally on localhost.

In general, the `verifier_id` is not trivial like the `default` and should be kept secret from other verifiers to avoid a spoofing attack. However, the `pod_id` is only known to the Kubernetes control plane and the verifier responsible for attesting to a pod, which makes the same attack harder to execute.

Now the Python script internally checks for the existence of the `verifier_id` and whether this identifier is associated with the ability to verify the specified group of pods. Once the checks have been passed, the agent sends a POST request each time to inform the master that a new offer has been generated. For this reason, there is a first behavioural distinction regarding the agent, namely, whether the Python script is running or not. The agent sends a POST request to an API `/collect` exposed by the master node on port 5002, different from the `/verify` API exposed on port 5000. If the script collects the request from the agent, the master will return a log like this:

```
192.168.0.102 - - [22/Apr/2025 15:06:09] "POST /collect HTTP/1.1" 200 -
```

And on the agent side:

```
INFO  keylime_agent::quotes_handler > POST request to /collect succeeded
```

On the other hand, if the script is not running, the master cannot collect the quotes, and the agent will throw an error:

```
ERROR keylime_agent::quotes_handler > Failed to send POST request to /collect...
```

Finally, the script waits to receive all the quotes requested for the pods under observation. Once all the quotes have been received, the script serialises the measurement list separately and sends it back to the Verifier to validate the system integrity.

The cases dealt with in the functional tests include:

- test with a correct Verifier permission
- test with an unrecognised Verifier
- test with an unrecognised pod for a given Verifier

9.2.1 Test with a correct Verifier permission

This section reports on successful tests that have passed both checks. First, the Verifier runs the above command on the terminal with its unique `verifier_id` (the test uses “default”, but this can be changed) to attest a specific Pod with `616ac316-caa8-46e8-b42b-dabd554c59cc` as `pod_id`:

```
curl -X POST http://localhost:5000/verify -H "Content-Type: application/json" -d ’{
    "verifier_id": "default",
    "pods": [
        {
            "pod_id": "616ac316-caa8-46e8-b42b-dabd554c59cc",
            "whitelist": [],
            "exclude_list": []
        }
    ]
}’
```

As this verifier is already registered in the local database `verifier_pod_mapping.db` managed with `sqlite`, and the pod identifier requested in the command is associated with it, the positive response given to the verifier corresponds to this:

```
{
    "master_node_response": {
        "aggregated_response": {
            "code": 200,
            "results": {
                "quotes": [
                    {
                        "details": "Pod attestation completed successfully",
                        "pod_id": "616ac316-caa8-46e8-b42b-dabd554c59cc",
                        "status": "success"
                    }
                ],
                "uuid": "default"
            },
            "status": "OK"
        },
        "message": "Response forwarded"
    },
    "message": "Verifier authorized (Verification initiated)"
}
```

9.2.2 Attestation test with an unrecognised Verifier

This attempt tries to verify the same previous pod but with another verifier identifier (`other_verifier_id`), the command launched is:

```
curl -X POST http://localhost:5000/verify -H "Content-Type: application/json" -d ’{
    "verifier_id": "other_verifier_id",
    "pods": [
        {
            "pod_id": "616ac316-caa8-46e8-b42b-dabd554c59cc",
            "whitelist": [],
            "exclude_list": []
        }
    ]
}’
```

As the Verifier is not registered on the system, the following message is returned to the Verifier and the attestation attempt is aborted:

```
{
    "message": "Unauthorized Verifier"
}
```

9.2.3 Attestation test with an unrecognised pod for a given Verifier

The last attempt will try to verify another pod with `72a2e8ee-d491-4c96-807b-1418ce611f0` as identifier, but with a Verifier identifier (`default`) already registered in the system:

```
curl -X POST http://localhost:5000/verify -H "Content-Type: application/json" -d ’{
    "verifier_id": "default",
    "pods": [
        {
            "pod_id": "72a2e8ee-d491-4c96-807b-1418ce611f0",
            "whitelist": [],
            "exclude_list": []
        }
    ]
}’
```

As the Verifier is logged on to the system but does not have permission to attest the specified Pod, the following message is returned to the Verifier and the attestation attempt is cancelled:

```
{
    "message": "Verifier not authorized to attest some pods"
}
```

9.3 Comparison with standard Keylime

Another enforced test concerns the duration of a remote attestation cycle of the Keylime framework, integrated into the proposed solution of Kubernetes, compared with the traditional Keylime framework. Traditional means without the Kubernetes environment, always with the four essential services running on the two machines, as described in the testbed section. In the proposed architecture, it is necessary to consider some times that are not usually present, such as the

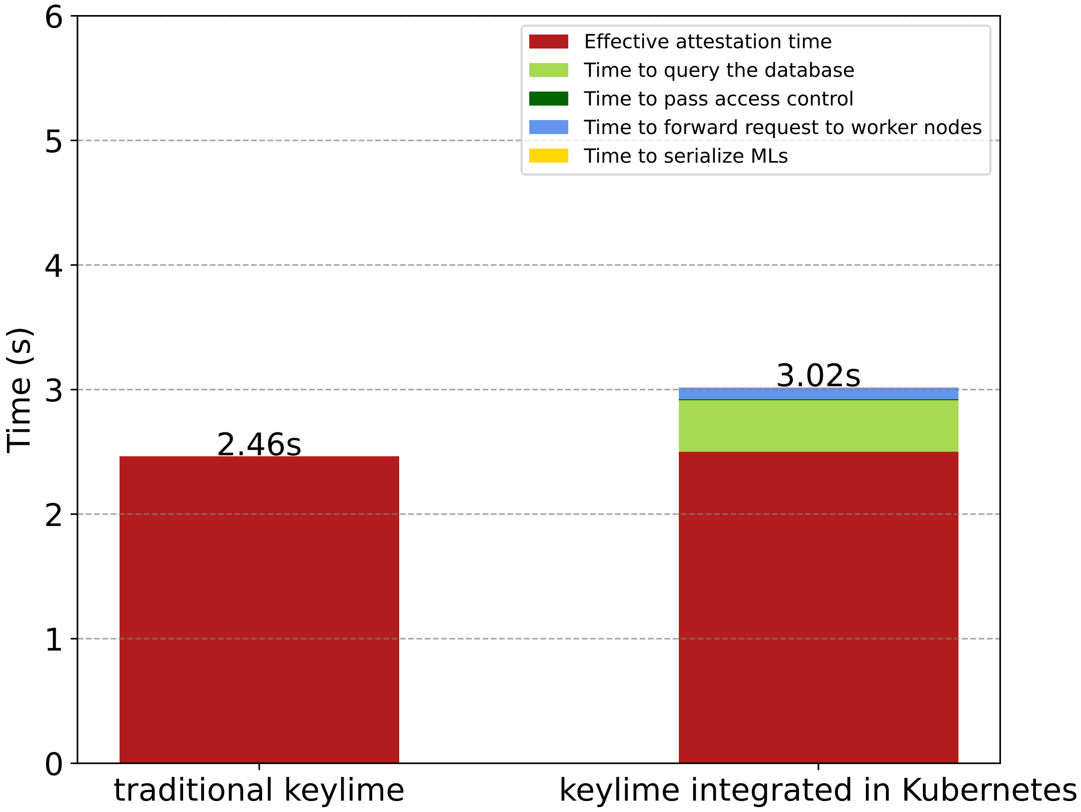


Figure 9.1. RA duration evaluated by increasing the number of pods

time to query the database to check the verifier's authorisation or the time to serialise all the measurement lists of different worker nodes to return to the right verifier.

As demonstrated in Figure 9.1, a comparison is provided between traditional Keylime and its integration within Kubernetes, focusing on the total attestation duration. The attestation process is comprised of multiple steps, including database querying, access control, forwarding of requests to worker nodes, and serialisation of measurement lists. Each of these steps contributes to the overall attestation time. The attestation duration in the traditional Keylime setup is measured at 2.46 seconds, while the Kubernetes-integrated version takes slightly longer at 3.02 seconds. This suggests that while Kubernetes adds some overhead, likely due to additional layers of orchestration, it remains within a reasonable range for attestation processes. The observed discrepancy can be attributed to factors such as the time required to query the database to verify the verifier's authorisation or the additional processing demands of managing distributed workloads. Despite the added delay, integrating Keylime with the proposed architecture could offer benefits such as access control features to limit verifier authorisation or enhanced management in a quote response. If security attestation remains effective within this additional time frame, the trade-off may be justified for environments where Kubernetes is the standard infrastructure.

9.4 Performance Tests

The metric utilised for the evaluation of performance was the duration required by the Keylime Verifier to execute an attestation cycle. This metric was determined by increasing the number of pods, commencing from 1 to 110. Performance beyond this threshold was not assessed, as Kubernetes imposes a default limit of 110 pods on standard cluster nodes. The number of pods was increased incrementally, beginning with two basic `nginx` and `apache` deployments, respectively, defined in yaml files as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apache-deployment
  labels:
    app: apache
spec:
  replicas: 1
  selector:
    matchLabels:
      app: apache
  template:
    metadata:
      labels:
        app: apache
    spec:
      containers:
      - name: apache
        image: httpd:latest
        ports:
        - containerPort: 80
```

The emphasis is placed on the segment of code specified as `replicas: 1`. Modifying the number of replicas for the deployment in question can increase the number of pods. These deployments are enforced by the following commands:

```
kubectl apply -f nginx.yaml
kubectl apply -f apache.yaml
```

Before describing the evaluation, it is imperative to taint the control plane master node with the following command:

```
kubectl taint nodes k8s-control node-role.kubernetes.io/control-plane-
```

to avoid deploying pods on it and trying to saturate all the resources on the worker under examination.

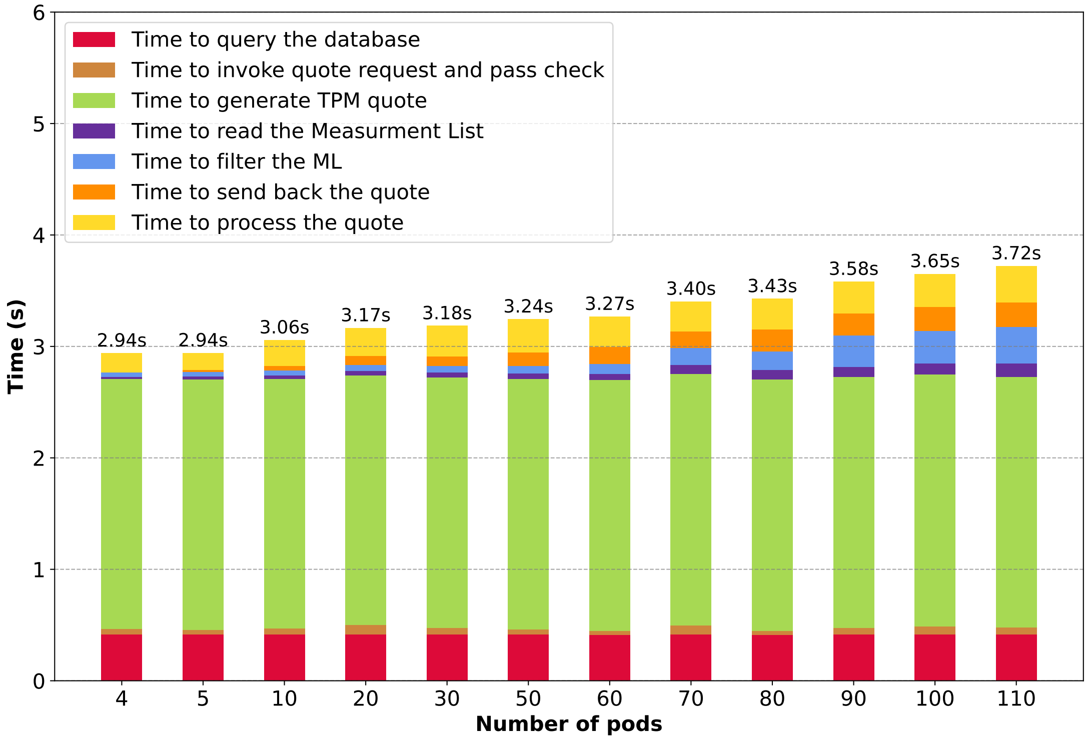


Figure 9.2. RA duration evaluated by increasing the number of pods

Analysis of figure 9.2 shows how the duration of the attestation process changes as the number of pods increases. The x-axis represents the number of pods, ranging from 4 to 110, while the y-axis measures the time in seconds. As each worker node has at least 3 management pods to join and properly run in the cluster, the test will start with the fourth pod deploying. The total duration of the attestation process consists of several stages, each of which contributes to the total time. These stages include querying the database, retrieving and verifying the offer request, generating the TPM quote, reading and filtering the measurement list, returning the quote and processing the received one.

As the number of pods increases, the overall attestation time also grows, though not in a straight line. In the beginning, with fewer pods, the time remains relatively stable at around 2.94 seconds. However, as the pod count rises, the attestation duration gradually increases, reaching approximately 3.72 seconds when there are 110 pods. This finding suggests that, while the process does scale with the number of pods, the increase in time is moderate and does not show exponential growth. The visual segmentation within each bar indicates that certain stages contribute more significantly to the total time than others. It appears that the time required to query the database and generate the TPM quote remains relatively consistent. The first procedure is essential to enforce fine-grained access control on the verifier applicant. In contrast, the second procedure is mandatory to read the content of the target PCR and build an integrity report to send back. As the number of pods increases, there is a slight increase in the time required to read and filter the Measurement List or to process the quote. However, the progression along these stages is influenced by the growth of the ML itself, with proportional growth as the size increases, contributing to the total escalation in attestation time. In other instances, the time required to invoke the request or to transmit the quote appears to be moderately extended. However, these delays are influenced by the erratic network traffic between the nodes, which is difficult to predict. This analysis suggests that while the attestation process does scale with system load, its efficiency is maintained within a reasonable range.

Chapter 10

Conclusions and future works

The pervasive implementation of cloud computing, notably in virtualised and multi-tenant environments administered by platforms such as Kubernetes, has precipitated a paradigm shift in IT infrastructure by virtue of the substantial flexibility and scalability advantages it provides. The proposed solution introduces new security features, particularly concerning workload integrity and sensitive data protection when running on shared or potentially untrusted infrastructure.

Despite the efficacy of RA, its implementation in multi-tenant Kubernetes environments gives rise to distinctive privacy concerns. As demonstrated in the relevant literature, the detailed integrity measurements (MLs) generated by IMA can contain information regarding the various components and applications operating on a shared node. This has the potential to divulge sensitive details about unrelated workloads or pods to verifiers. This predicament is especially salient in environments characterised by the presence of multiple independent verifiers, wherein the enforcement of precise access control over these logs poses a considerable challenge. Furthermore, the privacy issue depends on whether standard template hash fields within the MLs are predictable, which would allow for the offline deduction of measured components or configurations.

Although prior efforts in container attestation have made progress, privacy concerns have frequently been overlooked. Solutions such as Container-IMA and DIVE relied on TPM 1.2 and lacked robust support for modern container orchestration systems, such as Kubernetes pods. Recent approaches have adopted the use of TPM 2.0 and `cgroups` to establish a correlation between measurements and containers, thereby demonstrating enhanced integration potential. Despite the introduction of pod-aware templates and Keylime integration with Kubernetes Pod Attestation, challenges persist with multi-verifier scenarios and the potential for information leakage through template hashes.

The present thesis sets out a privacy-preserving remote attestation solution designed for multi-tenant Kubernetes environments with multiple verifiers. The new IMA template incorporates a nonce to prevent offline guess attacks, and it also supports the selective inclusion of log entries based on verifier authorisation. It is imperative to emphasise that solely pertinent attestation data is disseminated, thus ensuring the confidentiality of the cluster's internal architecture. The Kubernetes Master functions as a proxy to enforce access control and conceal pod locations. Concurrently, a modified Keylime Agent is employed to filter the measurement list before transmission.

The validity and performance of this solution were demonstrated through a series of tests. It has been demonstrated that the implementation of the authorisation logic has been executed correctly, as evidenced by the execution of functional tests. These tests have confirmed that the logic has been implemented in such a manner that it can only be invoked by authorised verifiers. Furthermore, these tests have also demonstrated that any attempts to invoke the logic by unauthorised parties are effectively blocked. A comparison of the solution with the standard Keylime framework demonstrated that integrating it into the Kubernetes architecture resulted in an acceptable level of overhead. The mean execution time for a standard Keylime RA cycle was found to be 2.46 seconds, whereas the Kubernetes-integrated solution required an average of 3.02 seconds. This discrepancy, amounting to 0.56 seconds, can be attributed to the implementation of enhanced security measures and centralised management processes, including database querying, access control checks, request forwarding, and Measurement List serialisation. Notwithstanding this marginal increase in time, the significant benefits gained in terms of privacy and access control in a Kubernetes environment justify it. The solution's scalability was confirmed through the conduct of performance tests, which involved an increase in the number of pods being attested. The total attestation time exhibited a moderate increase as the number of pods increased (from 2.94 seconds with a small number of pods to 3.72 seconds with 110 pods), without demonstrating exponential growth. The findings of the analysis demonstrated that the duration required for database querying and TPM quote generation exhibited relative constancy. Conversely, the time taken to read and filter the ML and process the quote increased in proportion to the log size. The findings indicate that the implemented modifications and architecture can effectively manage increasing loads without inducing unsustainable delays, thereby demonstrating a scalable solution that is mindful of resource consumption.

While the solution demonstrates strong performance and promising test results, it does have limitations that may affect future scalability. A salient concern pertains to its reliance on the Kubernetes Master node as a central proxy for authorization and request routing. This configuration could potentially result in a single point of failure or a bottleneck under conditions of heavy load or in highly distributed setups. It is recommended that future work explore the potential of distributed authorization mechanisms to enhance resilience. This exploration could encompass the execution of tests in larger and more complex Kubernetes environments. Furthermore, the integration of verifier-pod authorization with Kubernetes admission controllers for automation could be a fruitful avenue for future research. In the end, the refinement of the Master's state aggregation logic to enhance efficiency when handling responses from multiple Worker nodes is a promising area for further investigation.

Bibliography

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

[2] V. Singh and S. K. Peddoju, “Container-based microservice architecture for cloud applications”, International Conference on Computing, Communication and Automation (ICCCA), Greater Noida (India), May 2017, pp. 847–852, DOI 10.1109/CCAA.2017.8229914

[3] R. Morabito, J. Kjallman, and M. Komu, “Hypervisors vs. lightweight virtualization: A performance comparison”, 2015 IEEE International Conference on Cloud Engineering, Tempe (AZ, USA), March 2015, pp. 386–393, DOI 10.1109/IC2E.2015.74

[4] TCG working group, “Official TCG website”, https://trustedcomputinggroup.org/

[5] Trusted Computing Group, “TPM 2.0 Library specification: Architecture”, https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part1_Architecture_pub.pdf

[6] The Kubernetes Project, “Kubernetes Concepts”, https://kubernetes.io/

[7] A. Lioy, G. Ramunno, and D. Vernizzi, “Trusted-computing technologies for the protection of critical information systems”, International Workshop on Computational Intelligence in Security for Information Systems CISIS 2008, Genova (Italy), October 2008, pp. 449–455, DOI 10.1007/978-3-540-88181-0_10

[8] W. Artur, D. Challener, and K. Goldman, “A Pratical Guide to TPM 2.0”, Apress Open, January 2015, ISBN: 978-1-4302-6583-2

[9] W. D. Casper and S. M. Papa, “Root of Trust”, Encyclopedia of Cryptography and Security (H. C. A. van Tilborg and S. Jajodia, eds.), pp. 1057–1060, Springer, Boston, MA, 2011, DOI 10.1007/978-1-4419-5906-5_789

[10] Trusted Computing Group, “TCG Specification Architecture Overview”, https://trustedcomputinggroup.org/wp-content/uploads/TCG_1_4_Architecture_Overview.pdf

[11] Trusted Computing Group, “TCG PC Client Platform Firmware Profile Specification”, https://trustedcomputinggroup.org/wp-content/uploads/PC-Client-Platform-Firmware-Profile-Version-1.06-Revision-52_pub.pdf

[12] Trusted Computing Group, “TPM 2.0: A Brief Introduction”, https://trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-A-Brief-Introduction.pdf

[13] F. Bohling, T. Mueller, M. Eckel, and J. Lindemann, “Subverting Linux’ integrity measurement architecture”, 15th International Conference on Availability, Reliability and Security, Ireland, August 2020, pp. 1–10, DOI 10.1145/3407023.3407058

[14] S. Reiner, Z. Xiaolan, J. Trent, and v. D. Leendert, “Design and Implementation of a TCG-based Integrity Measurement Architecture”, 13th USENIX Security Symposium (USENIX Security 04), San Diego (CA, USA), August 2004, pp. 1–17

[15] The Linux Kernel documentation, “IMA Template Management Mechanism”, https://docs.kernel.org/security/IMA-templates.html

[16] N. D. Keni and A. Kak, “Adaptive containerization for microservices in distributed cloud systems”, IEEE 17th Annual Consumer Communications & Networking Conference (CCNC), Las Vegas (NV, USA), January 2020, pp. 1–6, DOI 10.1109/CCNC46108.2020.9045634

[17] IBM Cloud Education, “Docker”, https://www.ibm.com/topics/docker

[18] M. Hausenblas and S. Schimanski, “Programming kubernetes: Developing cloud-native applications”, O’Reilly Media, July 2019, ISBN: 978-1-492-04705-6

[19] N. Schear, P. T. Cable, T. M. Moyer, B. Richard, and R. Rudd, “Bootstrapping and maintaining trust in the cloud”, 32nd Annual Conference on Computer Security Applications, Los Angeles (CA, USA), December 2016, pp. 65–77, DOI 10.1145/2991079.2991104

[20] Keylime Project, “Official documentation”, https://keylime.readthedocs.io/en/latest/

[21] M. Souppaya, J. Morello, and K. Scarfone, “Application Container Security Guide”, NIST SP800-190, September 2017, DOI 10.6028/NIST.SP.800-190

[22] L. Wu, S. Qingni, X. Yutang, and W. Zhonghai, “Container-IMA: A privacy-preserving integrity measurement architecture for containers”, 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), Beijing (China), September 2019, pp. 487–500

[23] 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

[24] C. Piras, “TPM 2.0-based Attestation of a Kubernetes Cluster”, 2022, Politecnico di Torino

[25] Git Project, “Official Documentation”, https://git-scm.com/doc

[26] Calico Project, “Quickstart for Calico on Kubernetes”, https://docs.tigera.io/calico/latest/getting-started/kubernetes/quickstart

Appendix A

User Manual

This appendix describes how to configure the environment to test the proposed solution and provides the steps required to configure and deploy two nodes, an Attester and a Verifier. The Manual is based on a configuration in a VirtualBox environment and these two nodes must be able to communicate with each other over a local network. The Attester uses a TPM, which in VirtualBox is an emulated device, as opposed to a more serious configuration such as a software TPM emulator or even a real hardware TPM.

A.1 Attester machine

This section configures a host to be used as an attester in the remote attestation process. The machine must have a TPM 2.0. The operating system used for testing is Ubuntu Server 20.04 LTS, with a custom Linux kernel containing the IMA module modified as described in Appendix C.

A.1.1 Patching the Linux kernel

The first thing to do is install Git [25] and clone the git repository of the stable Linux kernel using the following command:

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

This will create the linux-stable directory. Change to the current directory and switch to a commit that permits applying the desired modification without error:

```
$ cd linux-stable
$ git checkout 88035e5694a86a7167d490bb95e9df97a9bb162b
```

The checkout command allows users to switch to a particular snapshot of the kernel, but it is also possible to verify that it is on one of the latest stable release branches (LTS) by wiping it:

```
$ git branch --contains 88035e5694a86a7167d490bb95e9df97a9bb162b
```

The reason for this checkout at the selected commit is that it has a custom IMA template with a template-hash field using a `SHA-256` digest rather than the actual `SHA-1`. Then, apply the patches provided with the thesis source code to obtain the custom Linux kernel:

```
$ git config --global user.email "you@example.com"
$ git config --global user.name "Your Name"


$ git apply /patches_dir/0001-ima_template_dep_cgn.patch
$ git apply /patches_dir/0002-ima_template_cgpath.patch
$ git apply /patches_dir/0003-ima_template_slt.patch
```

A.1.2 Preparing the kernel before compiling

There are a few steps to follow to proceed with compiling the kernel. First, install the dependencies:

```
$ sudo apt-get update
$ sudo apt-get install git fakeroot build-essential ncurses-dev xz-utils libssl-dev bc fle
```

Copy the current kernel configuration file to the linux-stable directory:

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

And then customize it:

```
$ make menuconfig
```

In the menu, select:

```
Security Options -> Integrity Measurement Architecture (IMA)
```

And then choose:

```
Default template                    -> ima-slt
Default integrity hash algorithm    -> SHA256
Default template hash algorithm     -> SHA256
```

Now, some options are needed to enable Docker Bridge and to ensure compatibility with Kubernetes:

```
Networking support -> Networking options ->
    <*> 802.1d Ethernet Bridging
    [*] Network packet filtering framework (Netfilter)

Networking support -> Networking options -> XDP sockets ->
    <*> XDP sockets: monitoring interface

Networking support -> Networking options -> Network packet filtering framework ->
    [*] Advanced netfilter configuration
    <*> Bridged IP/ARP packets filtering

Networking support -> Networking options -> Network packet filtering framework
-> IP: Netfilter Configuration ->
    <*> IP tables support (required for filtering/masq/NAT)
    <*> Packet filtering
    <*> Packet mangling
    <*> raw table support (required for NOTRACK/TRACE)
    <*> iptables NAT support
```

```
Networking support -> Networking options -> Network packet filtering framework
-> Core Netfilter Configuration ->
    <*> Netfilter nf_tables support
    <*> Netfilter connection tracking support
    <*> Network Address Translation support
    <*> MASQUERADE target support
    <*> "addrtype" address type match support
    <*> "conntrack" connection tracking match support
    <*> Netfilter Xtables support (required for ip_tables)
    <*> Netfilter nf_tables conntrack module
    <*> Netfilter nf_tables redirect support
    <*> Netfilter nf_tables nat module
    <*> Netfilter nf_tables reject support
    <*> REJECT target support
    <*> raw table support (required for TRACE)
    <*> "MARK" target support
    <*> "mark" match support

Networking support -> Networking options -> Network packet filtering framework
-> The IPv6 protocol -> IPv6: Netfilter Configuration ->
    <*> IP6 tables support (required for filtering)
    <*> ip6tables NAT support
    <*> MASQUERADE target support

Device Drivers -> Network device support ->
    <*> Universal TUN/TAP device driver support
    <*> Virtual ethernet pair device (veth)

Device Drivers -> Network device support -> Network core driver support ->
    <*> Virtual eXtensible Local Area Network (VXLAN)

File systems ->
    <*> Overlay filesystem support
```

Save changes and exit from the menu. The configuration file can now be opened:

```
$ sudo nano .config
```

To avoid compilation errors, comment out the following lines (in the Certificates for Signature Verification):

```
CONFIG_MODULE_SIG_KEY="certs/signing_key.pem"
CONFIG_SYSTEM_TRUSTED_KEYS="debian/canonical-certs.pem"
CONFIG_SYSTEM_REVOCATION_KEYS="debian/canonical-revoked-certs.pem"
```

After that, further flags are needed to check in order to ensure the correctness of the attester machine as a worker node of Kubernetes:

```
CONFIG_OVERLAY_FS=y
CONFIG_IMA_LSM_RULES=y
CONFIG_IP6_NF_FILTER=y
CONFIG_IP6_NF_MANGLE=y
CONFIG_IP6_NF_MATCH_HL=y
CONFIG_IP6_NF_MATCH_RPFILTER=y
CONFIG_IP6_NF_SECURITY=y
CONFIG_IP6_NF_TARGET_HL=y
```

```
CONFIG_IP6_NF_TARGET_REJECT=y
CONFIG_IP6_NF_TARGET_SYNPROXY=y
CONFIG_IP_NF_MATCH_RPFILTER=y
CONFIG_IP_NF_TARGET_MASQUERADE=y
CONFIG_IP_NF_TARGET_REDIRECT=y
CONFIG_IP_NF_TARGET_SYNPROXY=y
CONFIG_IP_SET=y
CONFIG_IP_SET_BITMAP_IP=y
CONFIG_IP_SET_BITMAP_IPMAC=y
CONFIG_IP_SET_BITMAP_PORT=y
CONFIG_IP_SET_HASH_IP=y
CONFIG_IP_SET_HASH_IPMAC=y
CONFIG_IP_SET_HASH_IPMARK=y
CONFIG_IP_SET_HASH_IPPORT=y
CONFIG_IP_SET_HASH_IPPORTIP=y
CONFIG_IP_SET_HASH_IPPORTNET=y
CONFIG_IP_SET_HASH_MAC=y
CONFIG_IP_SET_HASH_NET=y
CONFIG_IP_SET_HASH_NETIFACE=y
CONFIG_IP_SET_HASH_NETNET=y
CONFIG_IP_SET_HASH_NETPORT=y
CONFIG_IP_SET_HASH_NETPORTNET=y
CONFIG_IP_SET_LIST_SET=y
CONFIG_IP_SET_MAX=256
CONFIG_IP_VS=y
CONFIG_IP_VS_IPV6=y
CONFIG_NETFILTER_XT_MARK=y
CONFIG_NETFILTER_XT_MATCH_COMMENT=y
CONFIG_NETFILTER_XT_MATCH_MULTIPORT=y
CONFIG_NETFILTER_XT_MATCH_PHYSDEV=y
CONFIG_NETFILTER_XT_MATCH_RECENT=y
CONFIG_NETFILTER_XT_MATCH_STATISTIC=y
CONFIG_NETFILTER_XT_MATCH_TIME=y
CONFIG_NETFILTER_XT_MATCH_U32=y
CONFIG_NETFILTER_XT_SET=y
CONFIG_NETFILTER_XT_TARGET_NOTRACK=y
CONFIG_NETFILTER_XT_TARGET_REDIRECT=y
CONFIG_NETFILTER_XT_TARGET_TPROXY=y
CONFIG_NF_CONNTRACK_BRIDGE=y
CONFIG_NF_CONNTRACK_MARK=y
CONFIG_NF_NAT_REDIRECT=y
CONFIG_NF_REJECT_IPV4=y
CONFIG_NF_REJECT_IPV6=y
CONFIG_NF_TABLES_BRIDGE=y
CONFIG_NF_TPROXY_IPV6=y
CONFIG_NFT_COMPAT=y
CONFIG_SYN_COOKIES=y
```

Finally, compile the kernel and make it run faster:

```
$ sudo make localmodconfig
```

which will skip drivers that are not needed and make compiling faster.

A.1.3 Compiling and installing the new kernel

The following commands can be used to compile and install the kernel:

```
$ sudo make -j $(nproc)
$ sudo make modules
$ sudo make modules_install
$ sudo make install
```

the operation will take some time. After this, a system reboot is mandatory, but by default, GRUB boots the newly installed kernel. To manually select a kernel at startup, increase `GRUB_TIMEOUT` in `/etc/default/grub` to allow time for selection.

```
$ sudo nano /etc/default/grub
----
GRUB_TIMEOUT=10
#GRUB TIMEOUT STYLE=hidden
```

In this way the grub menu stuck in the selection phase for 10 seconds. After this run this command to update grub settings and then reboot the system:

```
$ sudo update-grub
$ reboot
```

Then add a custom IMA policy file called `/etc/ima/ima-policy`. The policies used for this thesis, taken from Keylime Documentation [20], are:

```
# PROC_SUPER_MAGIC
dont_measure fsmagic=0x9fa0
# SYSFS_MAGIC
dont_measure fsmagic=0x62656572
# DEBUGFS_MAGIC
dont_measure fsmagic=0x64626720
# TMPFS_MAGIC
dont_measure fsmagic=0x1021994
# DEVPTS_SUPER_MAGIC
dont_measure fsmagic=0x1cd1
# BINFMTFS_MAGIC
dont_measure fsmagic=0x42494e4d
# SECURITYFS_MAGIC
dont_measure fsmagic=0x73636673
# SELINUX_MAGIC
dont_measure fsmagic=0xf97cff8c
# SMACK_MAGIC
dont_measure fsmagic=0x43415d53
# CGROUP_SUPER_MAGIC
dont_measure fsmagic=0x27e0eb
# CGROUP2_SUPER_MAGIC
dont_measure fsmagic=0x63677270
# NSFS_MAGIC
dont_measure fsmagic=0x6e736673
# EFIVARFS_MAGIC
dont_measure fsmagic=0xde5e81e4

measure func=BPRM_CHECK mask=MAY_EXEC
```

Now it is time to reboot the system again; Check that the IMA module uses the `ima-slt` template by checking the IMA ML file:

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

A.1.4 TPM2 tools Installation

Install the following dependencies first

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

Install TPM 2.0 tools and TPM 2.0 Resource Manager packages:

```
$ sudo apt install tpm2-tools
$ sudo apt install tpm2-abrmd
```

Manually build and install libtss2 library:

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

In the VirtualBox environment shut down the VM and restart it with the flag TPM2 enabled:

```
Settings -> System -> TPM: switch from none to v2.0
```

in case of communication with real TPM, configure TPM Command Transmission Interface (TCTI) following the command in the Keylime documentation [20]. Start and enable the TPM 2.0 Resource Manager service:

```
$ systemctl start tpm2-abrmd.service
$ systemctl enable tpm2-abrmd.service
```

And check if it is working:

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

This should show “active (running)”. To also check that the tpm2-tools are working correctly, run the command:

```
$ sudo tpm2_pcrread
```

A.1.5 Keylime installation

For both Attester and Verifier nodes, clone first the Keylime repository from git:

```
$ git clone https://github.com/keylime/keylime.git
$ cd keylime
```

At this point, install Python’s virtual environment package, create a new virtual environment named ”.venv”, and activate it for isolated Python development:

```
$ sudo apt install python3-venv -y
$ python3 -m venv .venv
$ source .venv/bin/activate
```

and then install Keylime requirements and Keylime itself:

```
$ pip install . -r requirements.txt
$ sudo ./installer.sh
$ sudo python3 setup.py install
```

Now modify the endpoint configurations by exploring the file in the folder `/etc/keylime/` by saving and adjusting permissions if necessary:

```
$ sudo nano /etc/keylime/registrar.conf
...
ip = "192.168.0.113"
...

$ sudo nano /etc/keylime/verifier.conf
...
ip = "192.168.0.113"
registrar_ip = 192.168.0.113
...

$ sudo nano /etc/keylime/tenant.conf
...
verifier_ip = 192.168.0.113
registrar_ip = 192.168.0.113
...

$ sudo nano /etc/keylime/agent.conf
...
ip = "192.168.0.128"
contact_ip = "192.168.0.128"
registrar_ip = "192.168.0.113"
...
```

If the Keylime agent does not recognize the mTLS certificate from the tenant. Check if agents `trusted_client_ca` is configured correctly:

```
$ ls -l /var/lib/keylime/cv_ca/cacert.crt
$ sudo chmod 644 /var/lib/keylime/cv_ca/cacert.crt
$ sudo chown keylime:tss /var/lib/keylime/cv_ca/cacert.crt
$ sudo chmod 750 /var/lib/keylime/cv_ca
$ sudo chown keylime:tss /var/lib/keylime/cv_ca
```

If does not exist the certificate or even the folder create it with this keylime command, the password is usually located in the `/etc/keylime/ca.conf`:

```
$ sudo keylime_ca -c init -d /var/lib/keylime/cv_ca
```

Create the client and server certificates with the right password and set permissions:

```
$ sudo keylime_ca -c create -n client -d /var/lib/keylime/cv_ca
$ sudo keylime_ca -c create -n server -d /var/lib/keylime/cv_ca
$ sudo useradd -r -s /bin/false -g tss keylime
$ sudo chown -R keylime:tss /var/lib/keylime/cv_ca
```

In the end, you can start each service on a different shell (for the Agent, consult the next section):

```
$ keylime_registrar
$ keylime_verifier
$ keylime_tenant -c add --uuid d432fbb3-d2f1-4a97-9ef7-75bd81c00000 -f payload
```

where payload is a blank file and d432fbb3-d2f1-4a97-9ef7-75bd81c00000 is the default agent uuid.

A.1.6 Keylime Agent installation

Install first the following dependencies:

```
$ apt-get install libclang-dev libssl-dev libtss2-dev libzmq3-dev pkg-config -y
```

Since the last Agent release of Keylime is written in Rust, follow the on-screen instructions to install cargo packet management.

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Then export the PATH to use cargo everywhere in the filesystem:

```
$ export PATH="$HOME/.cargo/bin:$PATH"
```

Clone the repository of rust-keylime and test it:

```
$ git clone https://github.com/keylime/rust-keylime.git
$ cd rust-keylime
$ cargo test
```

A.1.7 Configuring Keylime Agent

Once the installation is completed, open the keylime-agent.conf file:

```
$ sudo nano keylime-agent.conf
```

In the [agent] section, set the binding ip address for the agent server, for example:

```
ip = "192.168.60.94"
```

Set the address where the verifier and tenant can connect to reach the agent, for example:

```
contact_ip = "192.168.60.94"
```

Finally, set the address of the registrar server with which the agent communicates.

```
registrar_ip = "192.168.60.94"
```

Do not change the port and the uuid, which will always be the same. To avoid error if /var/lib/keylime/agent_data.json is missing, create the file and set the following permission:

```
$ sudo touch /var/lib/keylime/agent_data.json
$ sudo chmod 660 /var/lib/keylime/agent_data.json
$ sudo chown keylime:tss /var/lib/keylime/agent_data.json
$ sudo chmod 750 /var/lib/keylime
$ sudo chown keylime:tss /var/lib/keylime
```

In this case, the Keylime agent does not recognize the mTLS certificate from the tenant, check, or even set these permissions:

```
$ ls -l /var/lib/keylime/cv_ca/cacert.crt
$ sudo chmod 644 /var/lib/keylime/cv_ca/cacert.crt
$ sudo chown keylime:tss /var/lib/keylime/cv_ca/cacert.crt
$ sudo chmod 750 /var/lib/keylime/cv_ca
$ sudo chown keylime:tss /var/lib/keylime/cv_ca
```

To run the Keylime Rust Agent, launch this command:

```
$ RUST_LOG=keylime_agent=trace cargo run --bin keylime_agent
```

A.1.8 Installing and configuring Kubernetes

The installation of Kubernetes is almost identical between the Attester and the Master node, and will be explained in the A.2.2 section as it is the most complete. For simplicity, the Keylime Verifier and all other attestation components are placed into the master node.

A.2 Verifier machine

A.2.1 Keylime Verifier and Registrar installation

This section describes the configuration of a separate machine to run the Keylime Verifier, Keylime Registrar, and Keylime Tenant components. The tests were performed with all three components running on the same machine acting as the Kubernetes master node, although they can be installed and run on separate machines. The design is always the same, where the master acts as a proxy exposing an API to the verifier, and the verifier does not know any information about the internal cluster because it talks only to the master.

A.2.2 Installing and configuring Kubernetes

This configuration is to be enforced for both the Master and the Attester node. First of all, set a mnemonic hostname, one for each machine:

```
$ hostnamectl set-hostname k8s-control
$ hostnamectl set-hostname k8s-agent
```

Execute this command to see the immediate changes:

```
$ bash
```

Then, on both nodes, configure the hosts in the network:

```
$ printf "\n192.168.60.93 k8s-control\n192.168.60.94 k8s-agent" >> /etc/hosts
```

These two nodes are reachable by configuring a NatNetwork in VirtualBox and enabling an additional network interface toward the local network, with a related `/etc/netplan/01-netcfg.yaml` config file for example:

```
network:
    ethernets:
        enp0s8:
            dhcp4: no
            addresses: [192.168.60.94/24]
            nameservers:
              addresses: [8.8.8.8, 8.8.4.4]
    version: 2
```

Setting, for instance `192.168.60.94` for the attester and `192.168.60.93` for the master. At this point, load the required modules and configure these files:

```
$ printf "overlay\nbr_netfilter\n" >> /etc/modules-load.d/containerd.conf
$ modprobe overlay
$ modprobe br_netfilter
$ printf "net.bridge.bridge-nf-call-iptables = 1\nnet.ipv4.ip_forward = 1\
\nnet.bridge.bridge-nf-call-ip6tables = 1\n" >> /etc/sysctl.d/99-kubernetes-cri.conf
$ sysctl --system
```

The last command is used to see if the sysctl configuration is up to date. Now, install and set up `containerd`, a container runtime, on a Linux system:

```
$ wget https://github.com/containerd/containerd/releases/download/v1.7.13/\
containerd-1.7.13-linux-amd64.tar.gz -P /tmp/
$ tar Cxzvf /usr/local /tmp/containerd-1.7.13-linux-amd64.tar.gz
$ wget https://raw.githubusercontent.com/containerd/containerd/main/\
containerd.service -P /etc/systemd/system/
$ systemctl daemon-reload
$ systemctl enable --now containerd
```

Then get `runc`, a container runtime the system needs to run to manage containers. Install it and set the permissions:

```
$ wget https://github.com/opencontainers/runc/releases/download/v1.1.12/runc.amd64 \
-P /tmp/
$ install -m 755 /tmp/runc.amd64 /usr/local/sbin/runc
```

It is also necessary to update the Container Network Interface (CNI) plugins, which are essential for managing networking in containerized environments like Kubernetes.

```
$ rm -rf /opt/cni/bin/*
$ wget https://github.com/containernetworking/plugins/releases/download/v1.4.0/cni-\
plugins-linux-amd64-v1.4.0.tgz -P /tmp/
$ mkdir -p /opt/cni/bin
$ tar Cxzvf /opt/cni/bin /tmp/cni-plugins-linux-amd64-v1.4.0.tgz
```

Now, take the classic `containerd` default configuration and fill a file with it. Locate this file in a new folder following these commands:

```
$ mkdir -p /etc/containerd
$ containerd config default > /etc/containerd/config.toml
```

At this point open the file `/etc/containerd/config.toml`:

```
$ nano /etc/containerd/config.toml
```

and manually edit and switch `SystemdCgroup` from false to true:

```
SystemdCgroup = true
```

Following the Kubernetes documentation [6], is also required to disable swap for the kubelet to work properly:

```
$ sudo swapoff -a
$ sudo sed -i '/ swap / s/^\(.*\)$/#\1/g' /etc/fstab
```

The last command makes this setting persistent. Now, set up a system to install Kubernetes packages, first installing the necessary dependencies. Secondly, download and store the Kubernetes package signing key and finally add a new APT repository for Kubernetes packages. The version used for Kubernetes is `v1.29` and the repository is set accordingly.

```
$ apt-get update
$ apt-get install -y apt-transport-https ca-certificates curl gpg
$ mkdir -p -m 755 /etc/apt/keyrings
$ curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.29/deb/Release.key | sudo gpg \
--dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
$ echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s\
.io/core:/stable:/v1.29/deb/ /' | sudo tee /etc/apt/sources.list.d/kubernetes.list
```

A the end of this process, it's time to update package lists again including the newly added Kubernetes repository, and reboot the system:

```
$ apt-get update
$ reboot
```

At this point is possible to install three essential Kubernetes tools.

```
$ apt-get install -y kubelet kubeadm kubectl
```

The first one, `kubelet`, runs on every node in a Kubernetes cluster and manages the lifecycle of the node. The second, `kubeadm`, is a tool for bootstrapping a cluster, and `kubectl` is the command line tool for managing each resource in a cluster. It is suggested that this command be run to prevent automatic upgrades that could cause version mismatches:

```
$ apt-mark hold kubelet kubeadm kubectl
```

where the version is decided when the repository is set up.

The following commands need to be executed **only** on the master node, as it is responsible for creating the cluster.

To effectively initialize the Kubernetes cluster, use this command:

```
$ sudo kubeadm init --pod-network-cidr=192.168.0.0/16 \
--apiserver-advertise-address=192.168.60.93 --control-plane-endpoint 192.168.60.93
```

where `192.168.60.93` is an example of a control-plane IP address. The command, in order, defines the pod's network range, assigns an IP address to the Kubernetes API server for future cluster communication, and establishes a fixed endpoint for the control plane.

After initializing a Kubernetes cluster, configure kubectl for the current user by granting access to a fresh Kubernetes admin configuration file:

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

To check if all pods are running at any time, use the following command:

```
$ kubectl get pods -A -o wide
```

where the `-A` shows the pod of all namespaces and `-o wide` show more details such as the worker's IP address where the pod is running.

Kubernetes does not provide built-in networking for Pods; it relies on a CNI (Container Network Interface) plug-in to handle networking between Pods and nodes. Calico is one of the most popular CNI, and following the official documentation [26] is installed with this procedure:

```
$ kubectl create -f https://raw.githubusercontent.com/projectcalico/\
calico/v3.29.2/manifests/tigera-operator.yaml
$ kubectl create -f https://raw.githubusercontent.com/projectcalico/\
calico/v3.29.2/manifests/custom-resources.yaml
```

where the first one installs the Tigera operator and custom resource definitions, instead, the second installs Calico by creating the necessary custom resource.

Finall,y the Kubernetes cluster is set up and it is necessary to join the required worker node to the cluster. Just execute this last command on the master node to create a token on which a reachable node can join the cluster as a worker node:

```
$ kubeadm token create --print-join-command
```

where the output will be a new command like `kubeadm join` with the required token and all the information to join. Run this command on the nodes that would become worker nodes in this cluster.

A.2.3 Proxy Kubernetes Master node

This section explains the YAML configurations required to implement the proxy feature in the proposed solution. This feature enables the Master node to act as an intermediary between external communications and the internal network for the Keylime Agent responsible for pod attestation.

The following YAML file defines a Deployment and a NodePort resource for a worker node (node1 in this example). The Deployment ensures that a pod is running on the specified worker node, while the NodePort Service exposes the pod on port 30080 of the node.

```
    apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment-node1
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: example
      node: worker1
  template:
    metadata:
      labels:
        app: example
        node: worker1
    spec:
      nodeSelector:
        kubernetes.io/hostname: k8s-agent
      containers:
      - name: example-container
        image: hashicorp/http-echo:0.2.3
        args:
        - "-text=Hello from worker1"
        ports:
        - containerPort: 5678
---
apiVersion: v1
kind: Service
metadata:
  name: example-service-node1
  namespace: default
spec:
  type: NodePort
  selector:
    app: example
    node: worker1
  ports:
  - protocol: TCP
    port: 80
    targetPort: 5678
    nodePort: 30080
```

On the other hand, another YAML file configures a deployment and a cluster IP service for the pod distribution service, which executes a Python script to determine the appropriate worker node for incoming requests. The deployment ensures that the pod runs on the control plane node and uses a custom container image that includes the Python script and all the necessary packages. The associated `ClusterIP` Service exposes the pod within the cluster on port 5000.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pod-distribution-deployment
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: pod-distribution
  template:
    metadata:
      labels:
        app: pod-distribution
    spec:
      nodeSelector:
        node-role.kubernetes.io/control-plane: "true"
      containers:
      - name: pod-distribution-container
        image: your-docker-repo/pod-distribution-service:latest
        env:
        - name: KUBERNETES_SERVICE_HOST
          value: "kubernetes.default.svc"
        - name: KUBERNETES_SERVICE_PORT
          value: "443"
        ports:
        - containerPort: 5000
        - containerPort: 5001
        - containerPort: 5002
---
apiVersion: v1
kind: Service
metadata:
  name: pod-distribution-service
  namespace: default
spec:
  selector:
    app: pod-distribution
  ports:
  - name: port-5000
    protocol: TCP
    port: 5000
    targetPort: 5000
  - name: port-5001
    protocol: TCP
    port: 5001
    targetPort: 5001
  - name: port-5002
    protocol: TCP
    port: 5002
    targetPort: 5002
```

In the end, the last YAML file defines an Ingress resource that acts as the system's reverse proxy. It manages incoming requests and forwards them to the correct worker node, applying the logic provided by the pod distribution service. The Ingress relies on the Pod Distribution Service to provide authentication and routing decisions, which are delivered by an ad hoc scheduled pod explained in the section A.2.4. Custom forwarding behaviour is implemented through annotations, and the Ingress rule outlines the path for the Keylime Agent API, as well as for the default backend service.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
  namespace: default
  annotations:
    nginx.ingress.kubernetes.io/auth-url: "http://pod-distribution-service.default
    .svc.cluster.local:5000/verify"
    nginx.ingress.kubernetes.io/auth-response-headers: "node_ip"
    nginx.ingress.kubernetes.io/configuration-snippet: |
      set $target "";
      if ($http_node_ip) {
        set $target "http://$http_node_ip:30080";
      }
    nginx.ingress.kubernetes.io/proxy-pass: "$target$request_uri"
spec:
  rules:
  - host: example.com
    http:
      paths:
      - path: /v2/agents/d432fbb3-d2f1-4a97-9ef7-75bd81c00000/
        pathType: Prefix
        backend:
          service:
            name: pod-distribution-service
            port:
              number: 5000
```

In a nutshell, with everything set up, these files create a setup where the Master node acts as a proxy, managing communication with the Keylime Agents on the Worker nodes and helping keep things secure and under control. Just make sure that the Pod Distribution Service has the correct permissions and RBAC roles to do its job. Also, confirm that your Ingress Controller is correctly installed and configured in the cluster to ensure everything runs smoothly.

A.2.4 Deploy Pod Distribution Service Pod

The subsequent section will provide a comprehensive guide to the deployment of the Pod Distribution Service Pod. Initially, it is necessary to create a Docker image containing the Python script and its dependencies. The `Dockerfile` provided with the thesis source code is employed, and the requirements are installed by following the command line log and activating a virtual environment. In the same folder, load the `Dockerfile` and the file which contains the logic of the distribution service pod service named “`pod_distribution_service.py`”, described in section B.4, and also provided with the thesis source code.

The `Dockerfile` must subsequently be located and loaded within the same folder as a separate file that contains the distribution service pod logic. This file is named “`pod_distribution_service.py`” which is described in the B.4 section of the Developer’s Manual and provided in the thesis source code.

Now you can build and push the image using the following commands:

```
$ docker build -t your-docker-repo/pod-distribution-service:latest
$ docker push your-docker-repo/pod-distribution-service:latest
```

When this procedure is completed correctly, apply the YAML configuration described before:

```
$ kubectl apply -f pod-distribution-service.yaml
```

Here, the deployment will be initiated with the requested pod, and the service resource will establish communication with all the necessary endpoints.

Appendix B

Developer Manual

B.1 IMA patch

This section provides the steps needed to create the IMA patch used in the proposed work with the help of the TORSEC research group of the Polytechnic of Turin.

B.1.1 `ima-slt` template implementation

The `ima-slt` template incorporates an additional field for the control group path that is not supported by iMA. The initial step in this process is to navigate to the directory containing the Linux kernel source code and modify the file `./security/integrity/ima/ima_template.c`. First add to the array called `supported_fields` the new content:

```
static const struct ima_template_field supported_fields[] = {
...
    {.field_id = "slt", .field_init = ima_eventslt_init,
    .field_show = ima_show_template_string},
};
```

In this code snippet, the following components are defined:

1. "`slt`" as the ima field identifier
2. "`ima_eventslt_init`" is responsible for initialising the field value for measurement events.
3. "`ima_show_template_string`" is responsible for writing the field value in the Measurement Logs

Subsequently, within the same file, a new built-in template must be defined named "`ima-slt`". The format string employed is "`dep|cg-path|d-ng|n-ng|slt`".

```
static struct ima_template_desc builtin_templates[] = {
    ...
    {.name = "ima-slt", .fmt = "dep|cg-path|d-ng|n-ng|slt"},
 };
```

Each string has a meaning, in particular: `dep` and `cg-path` fields are used respectively to denote the dependencies of the process that created the entry and the control group path needed to establish the pod UID. Both have been defined by the research group. While *d-ng* and *n-ng* are respectively the file digest and the file path. The term `slt` is the newly introduced field and represents a random nonce freshly generated at each measurement event recording. It is noted that the strings d-ng and n-ng represent the file digest and the file path, respectively.

Then it is necessary to insert the prototype of the new function ima_eventslt_init in the file ./security/integrity/ima/ima_template_lib.h

```
int ima_eventslt_init(struct ima_event_data *event_data,
                      struct ima_field_data *field_data);
```

and the related function is written in the file ./security/integrity/ima/ima_template_lib.c and reported here:

```
...
#include <uapi/linux/limits.h>
#include <linux/random.h>
#define NONCE_LEN 16
...
/*
* ima_eventslt_init - include an additional random field to make template
* hash unpredictable
*/
int ima_eventslt_init(struct ima_event_data *event_data, struct
        ima_field_data *field_data)
{
       char nonce[NONCE_LEN];
       char nonce_hex[(NONCE_LEN*2)+1];
       get_random_bytes(nonce, NONCE_LEN);
       for (size_t i = 0; i < NONCE_LEN; i++) {
               sprintf(nonce_hex + (i * 2), "%02x", nonce[i]);
       }
       nonce_hex[NONCE_LEN * 2] = '\0'; // Null-terminate the hex string
       return ima_write_template_field_data(nonce_hex, NONCE_LEN*2,
            DATA_FMT_STRING, field_data);
}
```

The function uses three local variables and a constant value:

- NONCE_LEN: a constant value has been set to 16 length characters, a sufficient length of random data to render the record unpredictable;
- **nonce**: is the buffer used for storing a random sequence of bytes of length NONCE_LEN;
- **nonce_hex**: represents the buffer for storing the hexadecimal representation of the nonce, with space allocated for an additional null terminator;

The function get_random_bytes() is invoked to populate the nonce buffer with a random sequence of bytes. The number of bytes to be generated is also specified by the constant. The following loop iterates over each byte in the nonce buffer, converting each byte into a two-character hexadecimal string and storing it in nonce_hex. This is since each byte (8 bits) is represented by two hexadecimal digits (4 bits each) to accurately depict its value. After the loop, nonce_hex is null-terminated to ensure it is treated as a valid string and, at the end, the function ima_write_template_field_data is called to write the nonce_hex string as template field data into the field_data parameter using DATA_FMT_STRING as data format. In a nutshell, the ima_eventslt_init function adds a random field to the template, making the template_hash unpredictable.

In the end, modify the ./security/integrity/ima/Kconfig file by adding the row with the "+" operator, which consent to add the ima-slt template to the Kernel menu configuration. In this way is possible to select this template as the default one before proceeding with the kernel compilation, through the make menuconfig command.

```
choice
        prompt "Default template"
        default IMA_NG_TEMPLATE
        ...
        config IMA_NG_TEMPLATE
                bool "ima-ng (default)"
        config IMA_SIG_TEMPLATE
                bool "ima-sig"
        config IMA_DEP_CGN_TEMPLATE
                bool "ima-dep-cgn"
        config IMA_CGPATH_TEMPLATE
                bool "ima-cgpath"
+       config IMA_SLT_TEMPLATE
+               bool "ima-slt"
endchoice

config IMA_DEFAULT_TEMPLATE
        string
        default "ima-ng" if IMA_NG_TEMPLATE
        default "ima-sig" if IMA_SIG_TEMPLATE
        default "ima-dep-cgn" if IMA_DEP_CGN_TEMPLATE
        default "ima-cgpath" if IMA_CGPATH_TEMPLATE
+       default "ima-slt" if IMA_SLT_TEMPLATE
choice
```

B.2 Filtering Measurement Log

The purpose of this section is to elucidate the code in order to filter an IMA measurement list, checking on entries related to a specific Kubernetes pod identified by pod_id. The interested file is the default one used to manage the identity and integrity report in the keylime rust agent: ./rust-keylime/keylime-agent/src/quotes_handler.rs. The filtering logic snippet of code is the following one:

```
async fn integrity(param: web::Query<Integ>, data: web::Data<QuoteData<'_>>,
 ) -> impl Responder {
 ...
let pattern = format!("/kubepods.slice/kubepods-besteffort.slice
            /kubepods-besteffort-pod{}{}", pod_id, ".slice");
let pattern_entry_pod = format!("/kubepods.slice/kubepods-besteffort.slice/
            kubepods-besteffort-pod");
let filtered_ima_measurement_list: Vec<String> =
    ima_measurement_list.clone().expect("REASON")
    .split('\n')
    .filter_map(|line| {
        let fields: Vec<&str> = line.split_whitespace().collect();
        if fields.len() >= 5 {
            let cgroup_path = fields[4];
            if cgroup_path.contains(&pattern_entry_pod) {
                if cgroup_path.contains(&pattern) {
                    return Some(line.to_string());
                } else {
                    return Some(fields[..2].join(" "));
                }
            } else {
                return Some(line.to_string());
            }
        }
        return Some(line.to_string());
    })
    .collect();
```

The code first constructs a specific formatted string pattern based on `pod_id` and to to identify relevant cgroup paths in the measurement list. This pattern is tailored for Kubernetes, but may need adjustment for other quick or lightweight orchestrators (i.e. k3s, minikube). Then the list is divided into lines by the `.split()` command and each line is evaluated individually:

1. In the event that the cgroup path corresponds to the designated pod pattern, the entirety of the line is preserved.
2. If there's a match with a broader pod pattern, only the first two fields are kept.
3. In the absence of such a measure, the entire line is retained.

The implementation of the filter is facilitated by the `filter_map()` function, which systematically processes each line. In accordance with the preceding logic, the function returns the entire entry or solely the initial two fields, as the verifier requires a pod belonging to a different tenant. At the end, horizontal filtered lines are collect into a new list, which can be further processed and returned to the verifier instead of the classic measurement list as follow:

```
let quote = KeylimeQuote {
    pubkey,
    ima_measurement_list,
    //ima_measurement_list,
    ima_measurement_list: Some(joined_list),
    mb_measurement_list,
    ima_measurement_list_entry,
    ..id_quote
};

...
let response = JsonWrapper::success(quote);
info!("GET integrity quote returning 200 response");
HttpResponse::Ok().json(response)
}
```

It should be noted that the code designated as `pod_id` is in fact that used by the verifier (though passed by the tenant), and it is this that is used to construct the request in the `./keylime/keylime/cloud_verifier_tornado.py` file in the following manner:

```
async def invoke_get_quote(
...
    res = tornado_requests.request(
    "GET",
    f"http://{agent['ip']}:{agent['port']}/v{agent['supported_version']}/quotes/integrity"
    f"?nonce={params['nonce']}&mask={params['mask']}"
    f"&partial={partial_req}&ima_ml_entry={params['ima_ml_entry']}"
    f"&pod_id={pod_id}",
    **kwargs,
    timeout=timeout,
)
```

B.3 Filtered ML verification

The Keylime Verifier has been modified to support the new version of the Measurement List provided in the Integrity Report. This section delineates the principal revisions and innovations introduced to align with the proposed format.

B.3.1 New ImaSlt Class

A new class ImaSlt has been introduced in the `keylime/ima/ast.py` file to handle the *ima-slt* template:

```
...
class ImaSlt(Mode):
"""
Class for "ima-slt". Contains the digest and a path.
"""
dep: Name       # dependencies
cg_name: Name   # cgroup-name
digest: Digest  # filedata-hash
path: Name      # filename-hint
nonce: Name     # nonce

def __init__(self, data: str):
    tokens = data.split(" ", maxsplit=4)
    if len(tokens) != 5:
        raise ParserError(f"Cannot create ImaSlt expected 5 tokens got: {len(tokens)}.")
    self.dep = Name(tokens[0])
    self.cg_name = Name(tokens[1])
    self.digest = Digest(tokens[2])
    self.path = Name(tokens[3])
    self.nonce = Name(tokens[4])

def bytes(self) -> bytes:
    return self.dep.struct() + self.cg_name.struct() + self.digest.struct() + self.path.st

def is_data_valid(self, validator: Validator) -> Failure:
    return validator.get_validator(type(self))(self.dep, self.cg_name, self.digest, self.p
```

It is the task of this novel class to manage the particular configuration of the ”ima-slt” template. This encompasses such elements as dependencies, cgroup-name, fieldata-hash, filename-hint, and nonce.

B.3.2 Modified Entry Class

The Entry class in `keylime/ima/ast.py` has been updated to support the new salted template format:

```
class Entry:
...
def __init__(
    self,
    data: str,
    salted_template: bool,
    validator: Optional[Validator] = None,
    ima_hash_alg: Hash = Hash.SHA1,
    pcr_hash_alg: Hash = Hash.SHA1,
):
    self.salted_template = salted_template
    self._validator = validator
    self._ima_hash_alg = ima_hash_alg
    self._pcr_hash_alg = pcr_hash_alg
    tokens = data.split(" ", maxsplit=3)
if len(tokens) != 4 and salted_template != True:
        raise ParserError(f"Cannot create Entry expected 4 tokens got:
        {len(tokens)}.")

    self.pcr = tokens[0]
    try:
        full_ima_th = tokens[1].split(':')
        self.ima_template_hash = bytes.fromhex(full_ima_th[1])
    except ValueError as e:
        raise ParserError(f"Cannot create Entry expected 4 tokens got:
        {len(tokens)}.") from e

    if salted_template == False:
        mode = self._mode_lookup.get(tokens[2], None)
        if mode is None:
            raise ParserError(f"No parser for mode {tokens[2]} implemented.")
    else:
        mode = self._mode_lookup.get("ima-slt", None)

    if salted_template == False:
        self.mode = mode(tokens[3])
        self._bytes = self.mode.bytes()
        self.pcr_template_hash = self._pcr_hash_alg.hash(self._bytes)
```

The Entry class now includes a salted_template flag to determine whether to use the new format or the old one.

B.3.3 Processing the Measurement List

The _process_measurement_list function is a significant component in the IMA validation process utilised by Keylime. The IMA measurement list is processed, with each entry being validated, and the running hash is computed, and the function has been updated to support both traditional and salted templates.

```
def _process_measurement_list(
    agentAttestState: AgentAttestState,
    lines: List[str],
    hash_alg: Hash,
    runtime_policy: Optional[RuntimePolicyType] = None,
    pcrval: Optional[str] = None,
    ima_keyrings: Optional[ImaKeyrings] = None,
    boot_aggregates: Optional[Dict[str, List[str]]] = None,
) -> Tuple[str, Failure]:
    failure = Failure(Component.IMA)
    running_hash = agentAttestState.get_pcr_state(config.IMA_PCR, hash_alg)
    assert running_hash

    # Check through the first line if is a salted template
    salted_template: bool = False
    if len(lines) > 0:
        tokens = lines[0].split(" ")
        if len(tokens) > 2:
            if tokens[2] == "ima-slt":
                salted_template = True
        elif len(tokens) == 2:
            salted_template = True
    ...

    for linenum, line in enumerate(lines):
        # remove only the newline character, as there can be the space
        # as the delimiter character followed by an empty field at the end
        line = line.strip("\n")
        if line == "":
            continue

        try:
            entry = ast.Entry(line, salted_template, ima_validator,
            ima_hash_alg=ima_log_hash_alg, pcr_hash_alg=hash_alg)

            # update hash
            if salted_template == False:
                running_hash = hash_alg.hash(running_hash +
                entry.pcr_template_hash)
            else:
                running_hash = hash_alg.hash(running_hash +
                entry.ima_template_hash)

            validation_failure = entry.invalid()
            if validation_failure:
                failure.merge(validation_failure)
                errors[type(entry.mode)] = errors.get(type(entry.mode), 0) + 1
```

The initial line of the measurement list must be examined in order to ascertain whether it corresponds to a salted template. In the event that this is the case, the `salted_template` flag must be set accordingly. It is then possible to update the `running_hash` based on the template type by iterating through each line in the measurement list. The `running_hash` is a cumulative hash value representing the system's integrity measurements, initialized with the PCR state from the agent's attestation state. Under the standard case for good practice, the `template_hash` value is recalculated over the fields on the right of the template descriptor. Alternatively, the `ima_template_hash` is used to extend the `running_hash`.

B.4 Pod Distribution Service script

The Pod Distribution Service script is responsible for the management of the verification process, the collection of responses, and the forwarding of aggregated results. The present script implements three discrete Flask applications, each of which is responsible for a different aspect of the pod distribution and verification process providing only relevant code (full script is in the thesis source code).

B.4.1 Kubernetes Node IP Retrieval

The `get_node_ip` function is responsible for interacting with the Kubernetes API to retrieve the node IP for a given pod UUID, with the purpose of mapping pods to their host nodes.

```
def get_node_ip(pod_uuid):
    config.load_kube_config()
    v1 = kubernetes.client.CoreV1Api()
    try:
        pods = v1.list_namespaced_pod(namespace="default")
        pod_name = next((pod.metadata.name for pod in pods.items if
        pod.metadata.uid == pod_uuid), None)

        if not pod_name:
            print(f"Pod with UUID {pod_uuid} not found")
            return None

        pod = v1.read_namespaced_pod(name=pod_name, namespace="default")
        node_name = pod.spec.node_name
        node = v1.read_node(name=node_name)
        return next((address.address for address in node.status.addresses if
        address.type == "InternalIP"), None)
    except ApiException as e:
        print(f"Exception when calling CoreV1Api->read_namespaced_pod: {e}")
    return None
```

This function uses the Kubernetes Python API to:

1. List all pods in the default namespace;
2. find the pod with the matching UUID;
3. retrieve the node information for that pod;
4. extract and return the internal IP address of the node.

B.4.2 Verifier and Pod Mapping Checks

The script incorporates specialised functions to verify the authenticity of verifier IDs and their authorisation to attest specific pods.

```
def check_verifier_id(verifier_id):
    conn = sqlite3.connect('verifier_pod_mapping.db')
    cursor = conn.cursor()
    cursor.execute("SELECT 1 FROM verifier_pod_mapping
            WHERE verifier_id=?", (verifier_id,))
    result = cursor.fetchone()
    conn.close()
    return result is not None

def check_verifier_pod_mapping(verifier_id, pod_id):
    conn = sqlite3.connect('verifier_pod_mapping.db')
    cursor = conn.cursor()
    cursor.execute("SELECT 1 FROM verifier_pod_mapping WHERE verifier_id=? AND
            pod_uuid=?", (verifier_id, pod_id))
    result = cursor.fetchone()
    conn.close()
    return result is not None
```

These functions interact with a SQLite database to perform security checks, ensuring that only authorised verifiers can attest to specific pods. The connection is invariably closed following the execution of the query, with a result always being provided.

Operations to install sqlite3 and create (and populate) the table are the following:

```
apt install sqlite3 -y
sqlite3 verifier_pod_mapping.db <<EOF
CREATE TABLE verifier_pod_mapping (
    verifier_id TEXT NOT NULL,
    pod_uuid TEXT NOT NULL,
    PRIMARY KEY (verifier_id, pod_uuid)
);
INSERT INTO verifier_pod_mapping (verifier_id, pod_uuid) VALUES ('abc', 'uuid1');
INSERT INTO verifier_pod_mapping (verifier_id, pod_uuid) VALUES ('def', 'uuid2');
INSERT INTO verifier_pod_mapping (verifier_id, pod_uuid) VALUES ('ghi', 'uuid3');
EOF
```

B.4.3 Multi-Application Concurrent Execution

The macro-steps of the Pod Distribution Service workflow are developed in the Verify function, which initiates the attestation process for multiple pods. At the same time, the Forward function consolidates the results and sends them to the Cloud Verifier, waiting for each agent to send a Collect notification. The script runs two separate Flask applications concurrently using a `ThreadPoolExecutor`:

```
...
from flask import Flask, request, jsonify

app_verify = Flask(__name__)
app_collect = Flask(__name__)
...
def run_app(app, port):
    app.run(host='0.0.0.0', port=port, threaded=True)

if __name__ == '__main__':
    with ThreadPoolExecutor(max_workers=3) as executor:
        executor.submit(run_app, app_verify, 5000)
        executor.submit(run_app, app_collect, 5001)
```

Adopting this approach enables the script to concurrently manage verification and collection processes across multiple ports and a separate thread that manages the forward part.

Verification Function

The verify function handles the initial verification request for pods. It's exposed as a POST endpoint at `/verify`:

```
responses = []
responses_lock = Lock()
all_responses_received = Event()
expected_responses = 0

@app_verify.route('/verify', methods=['POST'])
def verify():
    global expected_responses
    data = request.get_json()
    verifier_id = data.get('verifier_id')
    pods = data.get('pods', [])

    expected_responses = len(pods)
```

```
if not verifier_id or not check_verifier_id(verifier_id):
    return jsonify({"message": "Unauthorized"}), 401

# Start forward thread
threading.Thread(target=wait_and_forward_tcp, args=(
    verifier_id, expected_responses, all_responses_received,
        responses, responses_lock
)).start()

results = []

for pod in pods:
    pod_id = pod.get('pod_id')
    whitelist = pod.get('whitelist', [])
    exclude_list = pod.get('exclude_list', [])

    if pod_id:
        if not check_verifier_pod_mapping(verifier_id, pod_id):
            return jsonify({"message": "Not authorized to attest some pods"}), 401

        node_ip = get_node_ip(pod_id)

        if node_ip:
            # agent_ip = pod.get("agent_ip") not used in this context
            agent_port = pod.get("agent_port")
            agent_version = pod.get("agent_version", "2.2")

            nonce = data["nonce"]
            mask = data["mask"]
            partial = data.get("partial", "0")
            ima_ml_entry = data.get("ima_ml_entry", "0")

            agent_url = (
                f"http://{node_ip}:{agent_port}/v{agent_version}/quotes/integrity"
                f"?nonce={nonce}&mask={mask}&partial={partial}
                &ima_ml_entry={ima_ml_entry}&pod_id={pod_id}"
            )

            try:
                r = requests.get(agent_url, timeout=10)
                r.raise_for_status()
                quote_data = r.json()
            except requests.RequestException as e:
                quote_data = {"error": f"Failed to get quote: {e}"}

            results.append({
                "pod_id": pod_id,
                "node_ip": node_ip,
                "whitelist": whitelist,
                "exclude_list": exclude_list,
                "quote": quote_data
            })
        else:
            results.append({
                "pod_id": pod_id,
                "error": "Node IP not found"
            })
    else:
        results.append({
            "error": "Pod ID missing"
        })

return jsonify({"message": "Verification completed", "results": results}), 200
```

Systematically, the verification function performs:

1. Obtain and check the verifier's credentials and its authorization to attest specific pods.
2. For each pod in the request, it verifies the pod-verifier mapping and retrieves the node IP for the pod and sets a global variable `expected_responses` to the number of pods to attest.
3. It compiles results for each pod, including errors if any occur.
4. if checks are passed, send the integrity request to the right worker node by setting the url with the `node_ip` variable just retrieved.
5. It initiates the forwarding process by launching a separate thread to forward the quote back to the verifier. The *master-node-url* should be set to the actual IP and Master's port.

This function serves as the entry point for the verifier, sending the integrity request to the right worker node and setting up the necessary information for subsequent steps.

Concurrent Response Collection

The following code snippet implements a custom mechanism for concurrently collecting responses from worker nodes:

```
@app_collect.route('/collect', methods=['POST'])
def collect():
    global expected_responses
    data = request.get_json()

    # Extract quote from JSON payload, if present
    response_json = data.get("response")
    if not response_json:
        return jsonify({"error": "Missing 'response' field"}), 400

    with responses_lock:
        responses.append(response_json)
        if len(responses) == expected_responses:
            all_responses_received.set()

    return jsonify({"message": "Response collected"}), 200
```

This implementation uses threading primitives (`Locks` and `Events`) to safely collect responses from multiple worker nodes. The collection function is started together with the verification function, which signals the forward function when all the expected responses have been received.

Forwarding Operation

The forward function is responsible for aggregating collected responses and forwarding them to the cloud verifier. It's exposed as a POST endpoint at `/forward`:

```
def wait_and_forward_tcp(verifier_id, expected_responses,
    all_responses_received, responses, responses_lock):
    print(f"Wainting for {expected_responses} response...")

    if not all_responses_received.wait(timeout=10):
        print("Timeout waiting for all response.")
        return

    with responses_lock:
        if not responses:
            print("No valid response.")
            return

        quote_response = responses[0]  # full response JSON from Rust
        responses.clear()
        all_responses_received.clear()

    host = "localhost"  # Assuming the verifier TCP server is running on localhost
    port = 9999

    try:
        with socket.create_connection((host, port), timeout=5) as sock:
            data = json.dumps(quote_response).encode()
            sock.sendall(data)
            print("Quote send to the verifier by TCP")

            response = sock.recv(4096)
            print("Response from verifier:", response.decode())

    except Exception as e:
        print(f"Error to send: {e}")
```

This code checks for the presence of a verifier ID and the expected number of responses. Using the `all_responses_received` event, it waits for all expected responses to be collected and then sends a single aggregated response to the Cloud Verifier. The verifier receives the quote, without exposing any API, but listening to the response on a specific port through a script that runs a custom TCP server (script provided in the thesis source code under the name `quote_tcp_server.py`).

B.5 Authorised workload deployment

This section describes how workloads are deployed with verifier-based authorisation using a custom Python script that replaces the standard `kubectl apply -f command`.

The Usage of the new proposed command is the following:

```
$ python3 kubectl_apply.py -f <deployment.yaml> -verifier <verifierID>
```

The system checks the mapping `verifierID` to `podUUID` before forwarding any attestation request to check first if the Verifier is authorised to attest specific pods. If the mapping is missing, the request is blocked.

```
def get_pod_uuids(deployment_name):
    result = subprocess.run(['kubectl', 'get', 'pods', '-l',
        f'app={deployment_name}', '-o', 'json'], capture_output=True, text=True)
    pods = json.loads(result.stdout)
    pod_uuids = [pod['metadata']['uid'] for pod in pods.get('items', [])]
    return pod_uuids

def insert_verifier_pod_mapping(verifier_id, pod_uuids):
    conn = sqlite3.connect('verifier_pod_mapping.db')
    cursor = conn.cursor()
    for pod_uuid in pod_uuids:
        cursor.execute('INSERT INTO verifier_pod_mapping (verifier_id, pod_uuid)
                VALUES (?, ?)', (verifier_id, pod_uuid))
    conn.commit()
    conn.close()

def main():
    ...
    deployment_file = args.file
    verifier_id = args.verifier_id
    subprocess.run(['kubectl', 'apply', '-f', deployment_file], check=True)

    with open(deployment_file, 'r') as file:
        deployment = yaml.safe_load(file)
        deployment_name = deployment['spec']['selector']['matchLabels']['app']

    pod_uuids = get_pod_uuids(deployment_name)
    insert_verifier_pod_mapping(verifier_id, pod_uuids)
```

First, the script performs a classic deployment of the workload using the `kubectl apply` command and then it extracts the app label from the YAML file to identify the newly scheduled pods. It then retrieves pod UUIDs via the `kubectl` command and inserts (`verifierID`, `podUUID`) entries into the authorisation database, recording them to enforce access control for future verification requests.