POLITECNICO DI TORINO

Master's degree course in Computer Engineering

Master's Degree Thesis

TPM 2.0-based Attestation of a Kubernetes Cluster

Supervisors
Prof. Antonio Lioy
Dr. Ignazio Pedone
Dr. Silvia Sisinni

Candidate
Chiara Piras

Academic Year 2021-2022

Summary

The increasing adoption of the Cloud Computing paradigm made Kubernetes the de facto standard for most service providers. Kubernetes is an open source orchestrator platform to easily coordinate, manage and scale containerized workloads and services running in pods. A pod is a wrapper containing one or more tied containers designed to collaborate in pursuing a common goal. Since the pod represents the smallest scheduling unit, its integrity verification becomes necessary, to react fast to certain types of tampering, attack, or unexpected execution on a cluster node. The importance of such verification relies upon the final user concernment, which cannot take security assurance for granted: another user of the Cloud, an attacker, or even the Cloud provider itself, can gain access to the nodes and pods running its applications. Moreover, the COVID-19 pandemic drastically changed how people live and work, increasing, even more, the use of the Cloud infrastructure and consequently the number of cyber attacks. The technique exploited to perform a trustworthiness analysis of a physical node is called Remote Attestation, a process in which the job is delegated to an external (i.e. remote) entity commonly known as the verifier. Remote Attestation works well to validate the integrity of physical systems and nowadays it is a well-established technique. However, focusing the attention on containers, and more in general on pods, this process still possesses criticalities and open challenges. Several solutions have been proposed in recent years, but they are limited because either they rely on an outdated Trusted Platform Module version, such as DIVE and Container-IMA, or they are specific to a single container runtime, such as the Docker Container Attestation. This thesis work exposes a new solution to address exactly the continuous remote attestation of pods, meaning that the integrity of the underlying host and each pod running within it is periodically checked and validated, regardless of the container runtime used. The main goal reached is the possibility to detect rapidly unrecognized executions, intrusions, or tampering in a pod: whenever this happens, the pod is terminated and rescheduled with a fresh and uncompromised version, without affecting the integrity of the whole node, hence without rebooting the system and guaranteeing service continuity. Furthermore, the presented solution can be easily integrated into the Kubernetes control plane, which makes it a valuable starting point for future work. The solution proposes an attestation framework Trusted

Computing Group compliant, that relies upon the Trusted Platform Module 2.0, and it uses a custom version of the Linux ”Integrity Measurement Architecture” module.

Contents

Chapter 1

Introduction

Over the last few years, there has been an exponential increase in Cloud Computing adoption, thanks to its numerous advantages. Companies stop buying their own servers and start migrating towards a new form of elasticity where they pay and afford resources just according to their needs, getting more flexibility and reducing the associated cost. Theoretically, these resources are unlimited, they are always on, any time and any place, increasing the reliability by replicating data across different servers [1]. As a consequence of the Internet of Things (IoTs) diffusion, also Edge and Fog Computing paradigms start to be employed to cooperate with Cloud Computing. IoTs are basic devices that transmit over the internet a huge amount of sensor-collected data for specific purposes. If this data is directly transmitted to the Cloud Infrastructure, it can potentially lead to an overload of the network. For this reason, the data is first processed almost locally on edge nodes, which are physically placed near the devices. Then, the data is sent to the Cloud Infrastructure through fog nodes, which perform a second analysis and act as a bridge between the two. In this way, also IoTs can take advantage of the Cloud Computing model, with the real-time data analysis they require and without massive overhead.

On one hand, these solutions based on the "Infrastructure as a Service" model (IaaS) and the microservices architectural pattern bring a lot of benefits, but on the other one, they represent a new challenge from the security perspective. Final users, which in this context are commonly known as *tenants*, are concerned about taking security assurance from their cloud provider at face value [2]. In this new scenario, the applications are not executed anymore in the hardware kept by the final user, hence it is possible that a third party (i.e. other tenants, attackers, or even the cloud provider itself) gains easily direct access to private data. According to the 2022 Verizon Data Breach Investigations Report, system intrusion breaches are growing through stolen credentials four times more likely than exploiting vulnerabilities [3].

The lightweight virtualization technique offers containers as the standard way for deploying software in the cloud. If a successful attack on a node takes place, either tampering with the underlying kernel or with any container running on it, then it could be easily spread across different containers (i.e. applications) belonging to different tenants. The attack surface of container technologies is made large by the variety of core components involved: images, registries, orchestrators, containers, and the host operating system itself [4]. As an example, images could contain vulnerabilities and the secrets hard-coded within them can be discovered by malicious users. Moreover, even though this year misconfigured cloud storage has seen a slight leveling out, it still represents a dominant trend [3]. In addition, the minimal nature of IoTs makes them less protected and the perfect target for malware injections. It is crucial to emphasize that container adoption should be always coupled with security best practices and a proper orchestration system because even if originally a container is assumed to be *trusted*, it could be the target of any kind of software attack at any time.

For the reasons mentioned above, the run-time integrity verification of the whole cloud environment began a fundamental process, making the Trusted Computing (TC) technology a hot topic. Trusted Computing is based on a hardware solution that consent the remote attestation process, offering a *hardware root of trust* to protect the computing infrastructure and billions of endpoints [5]. The organization in charge to develop and maintaining the tools and documentation related to the TC technology is the Trusted Computing Group (TGC), which has established the Trusted Platform Module (TPM) chip as its core component.

Nowadays Kubernetes has become widespread, due to its open-source nature and as a consequence of the benefits introduced by Cloud Computing, becoming the de-facto standard for the orchestration technology. Within Kubernetes, the ”container” concept is replaced by means of ”pod”, the smallest scheduling unit, which can be composed of one or more containers. Datadog's 2021 Container Report shows that over the past year the number of pods per organization has doubled [6], underling the impressive growth that Kubernetes is having.

This thesis work aimed to address the problem of pod integrity verification, based on the previously proposed solutions. In particular, a new Integrity Measurement Architecture (IMA) template has been developed to allow discrimination between measurements entry produced by the host system and by a specific pod. This was done by exploiting the control group path of the process that generates the entry, which contains the pod ID. The Keylime framework has been used for the remote attestation, and it has been extended to provide the related functionalities. Specifically, it was made able to understand the new IMA template and was provided with new REST APIs. Finally, the solution has been validated with a testbed composed of one verifier machine and two attesters.

The thesis's structure follows this schema. The content of chapter 2 focuses on the concept of microservices and on the state of the art of the most used container runtimes, providing also an overview of the Kubernetes architecture. Chapter 3 provides an overview of Trusted Computing, the TPM 2.0 as its anchor technology, and the Remote Attestation process. The starting point of the proposed solution is underlined in chapter 4, with the explanation of three solutions developed for implementing the remote attestation of containers. Chapters 5 and 6 represent the core of the thesis work, with the design and the implementation of the solution. Finally, chapter 7 provides the test and validation phase, while chapter 8 contains the conclusions and future work.

Chapter 2

Kubernetes and Container Orchestration

Currently, the concept of monolithic application has been exceeded by the one offered by microservices: while a monolithic application is self-contained and provides itself the steps needed to achieve a particular goal, in the microservices paradigm the application is split into smaller pieces that have to cooperate all together to offer the whole service. Inspired by the successful adoption of tech giants, also many small companies and startups begin to adopt microservices as a game changer [7]. From the software engineering perspective, the main problems which have been addressed with this transition are related to modularity, scalability, maintainability, and fault tolerance of deployed applications [7], since now they can be seen as a set of separate executable files.

Microservices are weakly associated with each other to guarantee that a change in a single microservice has a poor impact on another one. To obtain such an isolation degree, many researchers evaluated different virtualization techniques through performance analysis, pointing out that the best choice was container exploitation [8]. Containers, entities that emerged with the lightweight virtualization technique, provide an Operating System-level virtualization method with the same properties (such as scalability, elasticity, and isolation) offered by full virtualization, without the overhead and the complexity associated with it. For instance, they provide higher performance, less memory requirement, and a cheaper infrastructure cost [8].

In a Cloud Computing scenario, where hundreds or thousands of applications are running, some automated system is needed to establish in which physical machine of the cluster a single container (a single microservice) should run, eventually with one or more replicas, and to monitor and control the whole containerized environment. This is where Kubernetes comes into play, an orchestration system that today is extensively adopted.

2.1 Microservices

In the microservices architecture, an application is divided into collaborating microservices, which is the idea behind the Cloud Computing model. This results in interesting advantages [9]:

- *Technology heterogeneity*: the possibility to adopt different programming languages for each microservice belonging to the application, guaranteeing great flexibility.
- *Resilience*: if one microservice fails then it does not affect the whole application. In this case, either the offered service is no longer available but the application is still running, or there is the chance to run multiple instances of the same microservice on different machines so that if a machine running the specific microservice is down, there is the assurance that the service is still guaranteed.
- *Scalability*: only the microservices that need actual scaling can be run in multiple instances. If a single microservice is CPU-consuming or its memory consumption is high, a common solution is to split the load between different machines and improve the computational power. Increasing or decreasing the number of instances of a given microservice is also called elastic deployment.
- *Ease of deployment*: given this new architectural model, a simplified software upgrade has been reached, thanks to the development agility which consent to easily switch from an old version of the software to a new one and vice versa.

On the contrary, from a disadvantageous point of view, the architectural pattern of microservices introduces additional difficulties, since scaling and monitoring of a microservices-based application is a more complex task than in the case of a monolithic one [7]. As an example, in case of a failure derived from a chain of calls to different microservices, ad-hoc debugging techniques are required. Then, the necessity to redesign the existing applications and develop the new ones in order to make them distributed, makes also observability and tracing difficult jobs, having independent components of the application running on different machines and exchanging messages through the network. Being the network significantly more involved in consent also the correspondence between microservices, new entry points of attack come up, giving birth to security criticalities in which a perimeter-focused solution is no longer possible. *"Defense in depth as a concept of placing multiple security mechanisms on different levels throughout the system becomes more feasible with microservices"* [10].

2.2 Container Runtimes

The common solution for creating microservices is the usage of containers. Containers represent a valuable way to package and run pieces of an application because

their nature allows to easily start, replace and isolate them, with the lowest overhead possible. Agile development, environmental consistency and compartmentalization, and the possibility to scale depending on the load at a given point in time are all benefits resulting from their adoption [4].
To be able to run containers in a host operating system a software component that can act as a container engine is needed, for instance, something which will be responsible for managing the life cycle of the containerized environment. Loading the images from a repository, creating and launching the containers themselves, and monitoring the available system resources, are all tasks of this container engine, which is commonly called "container runtime". While a container orchestrator such as Kubernetes is managing containers inside clusters (so containers that are running on different physical servers), a container runtime is managing the containers running on an individual physical machine. Of course, different container runtimes can cooperate and work together, so that you can have different container runtimes on different nodes of your cluster, and different container runtimes on the same node too.

2.2.1 Open Container Initiative

The Open Container Initiative (OCI) is a Linux Foundation Project developed in 2015 for creating container standards that currently provide two specifications. The first specification is called runtime-spec, and it is related to how to run a filesystem *bundle*, while the second specification is called image-spec and is related to how to get a filesystem *bundle* [11]. A bundle is basically a directory in the file system that wraps the container metadata, configuration data, and the root file system, and it is created by following the image-spec. So, the image-spec is specifying how to download and unpack in a bundle a container image together with its configuration data. Then, following the runtime-spec, such a bundle can be received by any OCI compliant runtime, which will be in charge to run it [12]. Actually, with the runtime-spec, an engine seed called **runc** has been implemented, as a common starting point in the container runtime development process. Additional functions can be added to runc, and it is commonly defined as a *low-level container runtime*. In other words, runc is defining the primitives that can be used to deal with containers (start, stop, destroy, and so on), and can be used internally to develop other container runtimes on top of it. On the other hand, systems that implement the image-spec and internally use runc, are commonly defined as *high-level container runtimes*. The importance of OCI introduction lies in the development of standard container runtimes by programmers. Consequently, it is possible to have containers able to run in any OCI-compatible container runtime.

The following paragraphs will provide a brief overview of the most used container runtimes, showing in the end a comparison between the trends of the last two years.

2.2.2 Docker

Architecture

Docker is an open-source platform that uses OS-level virtualization to deliver applications inside containers. Originally it was developed as a monolithic application, and after the introduction of the OCI standard, it has been decoupled into independent components, as shown in figure 2.1. In the original docker architecture, everything was done by the Docker Engine. Nowadays it is used only as a high-level component to initiate the container creation, while containerd and runc provide the actual container runtime functionality (respectively as image-spec and runtime-spec). The shim process is needed to decouple the container from the runtime, to avoid long-running runtime processes [12].

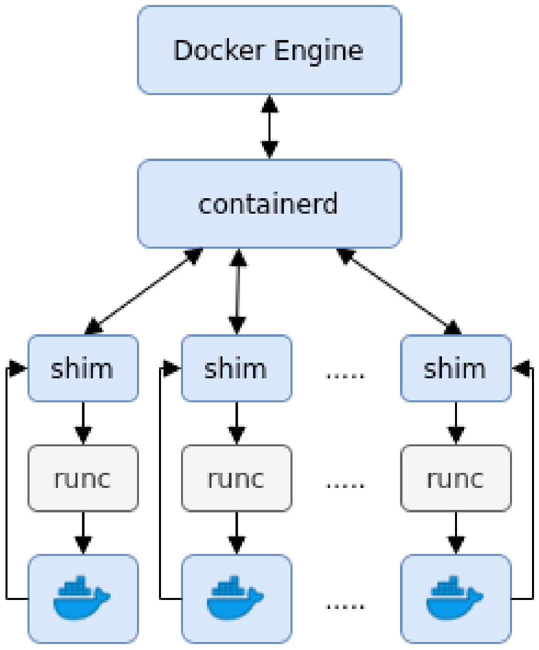


Figure 2.1. Docker components. (Source: [12]

Main Concepts

Containerd provides the management of the container's life cycle at a high level, dealing with the image recovery from a registry. In the case of Docker, typically the registry is *Docker Hub*, which offers read-only templates exploited to get a fully portable container. Containerd will be explained more in detail in section 2.2.3. Docker uses the concept of *automatic build*, where a *Dockerfile* is provided to build a single image and create one container. The content of a Dockerfile is similar to a recipe where a list of commands is issued in sequence and developers are free to use any combination of them.

Currently, starting from version 1.24 of Kubernetes (k8s) Docker Engine has been deprecated due to the presence of the Docker shim process, even though it was the first container runtime used. Containerization became an industry standard and the necessity to have multiple container runtimes and a Container Runtime Interface (CRI) to handle all of them emerged [13]. Nowadays Docker Engine can be used in Kubernetes only with the additional service cri-dockerd.

2.2.3 Containerd

Containerd is a high-level container runtime created by Docker, and for this reason, it has most of Docker's characteristics. Specifically, it is a daemon that runs in the background for handling each container's life cycle. Containerd internally uses runc as runtime-spec, hence it starts by pulling a container image from a repository, creates a bundle, and then it forwards the latter to runc. Runc can then start, stop, and destroy the container. Containerd is OCI-compliant, and after its creation, it was donated to the Cloud Native Computing Foundation as a basis for developing also new solutions [14]. A disadvantage of this container runtime is that being a daemon if it is restarted then all the containers managed by him will be restarted too.

2.2.4 Podman

Podman is an open-source, daemon-less, and Linux-native container engine developed by Red Hat. To be daemon-less means that there is no program in the background for handling processes since all the containers and pods are launched as child processes following the fork/exec model[15]. Podman is OCI-compliant and works with the pod concept: pod definitions can be directly exported to YAML files, guaranteeing compatibility with Kubernetes. Actually, Podman is one component in the containerization scenario, since it works together with other tools: Buildah which is used to build the containers, Skopeo which acts as an inspector for the images, runc to run the containers, and, optionally, crun to get more flexibility and control. An interesting feature is that Podman permits running rootless containers, which means building, launching, and maintaining containers without the need for root privileges on the host, and this is the case where the runtime crun can be useful. All the mentioned tools that cooperate with Podman work with any other OCI-compatible container engine, which makes very easy the transition from other container runtimes. Starting from June 8, 2022, Podman is available also for Windows systems [16].

2.2.5 CRI-O

CRI-O is an alternative container runtime OCI-compatible, specifically created to work with Kubernetes. When Kubernetes wants to launch a pod, it contacts the CRI-O daemon that first will retrieve the related image (or images) from a container registry. The latter operation is done by exploiting the *container/image* library, while to manage the images on disk CRI-O uses the *container/storage* library. Then, the daemon generates a JSON file with a description of how to run the containers. This file is forwarded by default to runc, which is in charge of actually launching the containers within the file, but this is not mandatory since CRI-O also supports other low-level OCI-compliant container runtimes [17]. CRI-O is considered an implementation of the Kubernetes CRI, explained more in detail in section 2.3.5. A benefit resulting in the usage of the CRI-O daemon is that when it is restarted, containers are not. This happens because actually there is another component, called "*conmon*", in charge to monitor all the containers.

On the other hand, CRI-O has no default tool for image management, and like Podman, it collaborates with Buildah or Skopeo.

2.2.6 Usage Trend

According to the 2021 Sysdig Container Security And Usage Report and as shown in figure figure 2.2, during the past year there has been a clear shift in choosing the container runtimes. Organizations start to move away from Docker in favor of newer and OCI-compliant container runtimes, such as containerd and CRI-O [18]. However, it is worth noting that Docker still being the most used container runtime among organizations, it dipped below 50% for the first time in five years only in 2022 [19], as shown in figure 2.3.

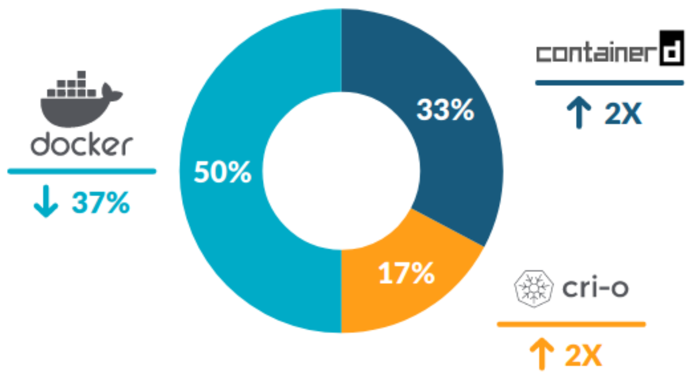


Figure 2.2. Container Runtimes Trend in 2021 (Source: [18])

2.3 Kubernetes

The deployment of containers requires some orchestration software to coordinate, manage and scale the total workload. Kubernetes is an open-source system initially developed by Google with this specific purpose. Today, according to the 2022 Sysdig Cloud-Native Security And Usage Report, if orchestrators based on Kubernetes like Red Hat OpenShift are included, Kubernetes is used 96% of the time [19]. In the following paragraphs, a brief overview will be presented.

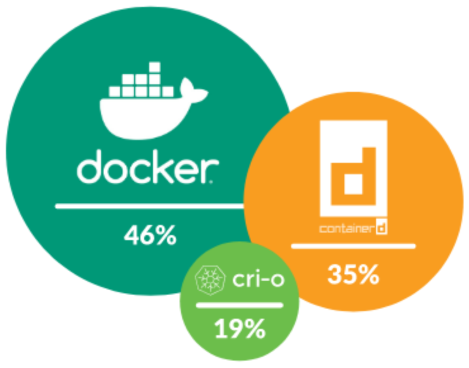


Figure 2.3. Container Runtimes Trend in 2022 (Source: [19])

2.3.1 Architecture

In the Kubernetes architecture, there is a distinction between master nodes and worker nodes. A master node is in charge of running an actual control plane to monitor and control the cluster, so it is responsible to manage the status of the *pods* that are currently running within the cluster. A pod is the smallest scheduling unit that can be handled by Kubernetes, a sort of wrapper which can be composed of one or more separate highly coupled containers. The usage of the pod concept brings the advantage of context sharing in terms of storage and network resources. On the contrary, a worker node is in charge of running the pods that are scheduled by the master, starting from *YAML configuration files*. YAML is a data serialization language that in this case is used to communicate new deployments to the orchestration system, as explained in section 2.3.2.

Master Node

A master node implements the control plane of the cluster, and being the general controller it should run in multiple instances to ensure availability and load balancing. Starting from the *desired state* specified by the tenant, a master node is continuously checking it against the *actual state* of the cluster. Each master node is composed of four main components [13]:

- **Kube API Server**: All the interactions with the cluster pass through the API server, which represents the entry point for monitoring, changing, and controlling the cluster status.
- **Kube Scheduler**: It is the component in charge of scheduling the pods of an application, by selecting the corresponding worker nodes based on the currently available resources (memory, CPU, and so on).
- **Kube Controller Manager**: It is the control loop logical component, which is in charge to watch the status of the cluster through controllers. Each controller is triggered and reacts by communicating with the kube API server when the actual state does not match the desired one.

- **Cloud Controller Manager**: It is a component used to handle specific cloud provider controllers used to allow communication between the cloud platform and the cluster.
- **Etcd**: It is the database of the cluster that provides consistency by exploiting the Raft Consensus Algorithm. It stores each desired state, to make Kubernetes intervene when any of them changes.

Worker Node

Worker nodes are responsible to execute the pods scheduled by the master, and they are composed of three key components [13]:

- **Kube-proxy**: It is the component responsible to handle the networking related to the *services* concept, explained in 2.3.3.
- **Kubelet**: It is the agent in charge to run and control the pods by interacting with the container engine and keeping the master node updated.
- **Container engine**: It is the container runtime that manages the execution of containers. A node can have multiple container engines: as stated in paragraph 2.3.5 Kubernetes supports different container runtimes as long as they are OCI-compatible.

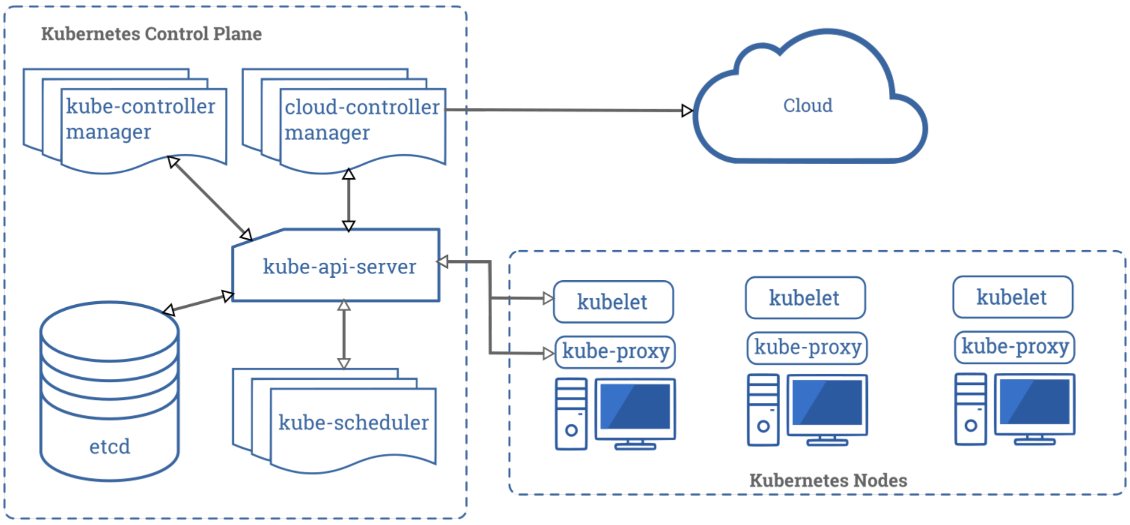


Figure 2.4. Kubernetes Architecture (Source: [13]

2.3.2 Workload Resources

Kubernetes offers a set of workload resources that can be used together to create and manage pods on behalf of the tenant. In the Kubernetes context, a workload is a running application for which the desired state has been specified. The job of workload resources is to configure the controllers handled by the kube controller manager in master nodes. In the following sections, the two main workload resources provided natively by Kubernetes will be presented. However, thanks to its extensible nature, Kubernetes permits to add a *custom resource definition* to get additional behaviors that are not part of Kubernetes' core [13].

Deployments

A deployment is a resource used to describe the desired state of an application. It packages all the related information within a YAML file which will be translated into API requests from the *kubectl* component. The kubectl is the Kubernetes command-line tool. In the deployment is provided the application life cycle, specifically, for each pod is present:

- the name, the image, and the port of each container that constitutes it, information that will be passed to the container engine component.
- the number of replicas that it should have, guaranteeing the horizontal scalability to facilitate more load. This information is managed internally by the ReplicaSet resource, as stated in 2.3.2.
- the way in which it should be updated. This consents to pause or continue one deployment and to perform a roll back to a previous version.

ReplicaSet

ReplicaSet is the Kubernetes resource in charge to replicate a given pod. It ensures that the set of replicas specified in the deployment is running at any time. This can be seen as three different tasks:

- Initially, when a new deployment is provided, ReplicaSet is responsible to instantiate the proper number of replicas for each pod.
- When any kind of error occurs in a pod, ReplicaSet is responsible to replace the entire set with a new one to be spawned.
- When the life cycle of a pod terminates, ReplicaSet is responsible to eliminate the entire set associated with it.

2.3.3 Service

Containers within a pod share the network namespace and for this reason, they see the same pod IP address. This results in easily enabling communication between each other via the loopback network interface. However, pods are non-permanent resources, meaning that they unpredictably change their IP address. Any time a pod is either updated or deleted and rescheduled, it changes its IP address. For this reason, a mechanism to establish how to connect and interact with the application was needed. Kubernetes services are an abstraction that solves this problem by assigning the workload pods to a logical set, which has a fixed IP address. In other words, services consent to expose applications executed in a set of running pods. The default option to expose is through *ClusterIP*, which makes the application reachable only within the cluster. If the application needs to be reachable from the outside, the solutions proposed by Kubernetes are the usage of a *load balancer* or a *node port* [13]. The latter requires knowing in advance which port to contact. Each pod is bound to its service through a selector field specified in the YAML file.

2.3.4 Network Plugins

In Kubernetes, the networking is managed by network plugins called Container Network Interface (CNI). While the kube-proxy component is responsible to reflect services by managing IP tables, the CNI plugins implement the overlay network. CNI plugins interact with the container runtimes any time a network operation is needed. Specifically, the CNI plugins follow this execution flow [20]:

- The container runtime needs a network operation so it calls the CNI plugin specifying the expected command to be executed.
- If it is already present a network configuration, it is sent from the container runtime to the CNI plugin, along with the needed container data.
- The CNI plugin executes the command and returns the result.

In the specific case of Kubernetes, the execution flow works in this way. When kubelet needs to make network operations on a pod, it does not interact directly with the CNI, rather it communicates with CRI and consequently with the container runtime. The container runtime is responsible to load the CNI plugin to implement the k8s network model. Then, the CNI is called twice: one to set up the loopback interface, and the other to set up the eth0 interface. It is possible to choose among different CNI plugins in the Kubernetes ecosystem, and when none of them is specified the default CNI plugin is noop [13].

2.3.5 Container Runtimes Interaction

In a deployment (i.e. in a YAML configuration file) are specified the container images that a pod should run, not the details related to *how* they should run. As a result, Kubernetes worker nodes need a container engine, as stated in 2.3.2. The agent able to communicate both with the control plane and with the specific container runtimes running on the node is Kubelet. Kubelet communicates with different container runtimes through the Container Runtime Interface (CRI). CRI is a protocol developed both to overcome the difficulties of integration between container runtimes and Kubernetes code and to specify the functions that a container runtime should have and perform on behalf of Kubernetes. CRI specification gives the possibility to communicate with all the container runtimes in a standardized way, without any hard-coded support and reliance on just one of them. The main functions that a container runtime should have according to the CRI can be summarized as follows [13]:

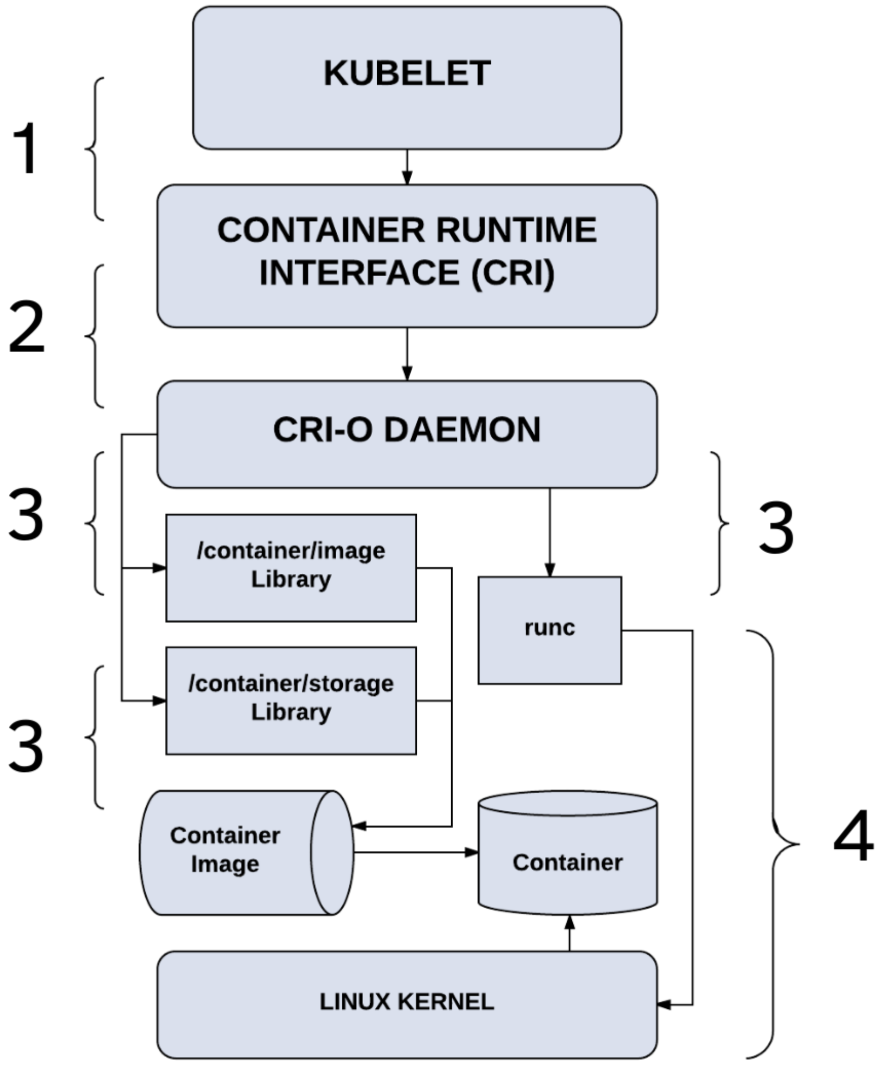


Figure 2.5. Kubernetes CRI interacting with CRI-O (Source: [21]

- Being able to start and stop a pod.
- Being capable to perform any operation to deal with a single container (start, pause, stop, delete, and kill).
- It should be able to pull the images from a registry and create the associated containers within a pod.
- It should supply the control plane with log files and metrics.

CRI supports only the container runtimes which are OCI-compliant, and from a technological point of view, it consists of protocol buffers and gRPC API.

As an example, in figure 2.5 can be seen the logical flow of interaction between CRI and the container runtime CRI-O:

1. When kubelet is asked to start a pod, the Container Runtime Interface is called.
2. CRI is configured to call the specific container runtime that is used in the worker node, that in this case is the CRI-O daemon.
3. CRI-O daemon, as stated in 2.2.5, first pulls the container images from a registry through the library *container/image*, then it stores them on disk through the library *container/storage*, and then calls runc to run the containers.
4. runc is responsible to contact the Linux Kernel which will start the container processes in the specified namespace or cgroup.

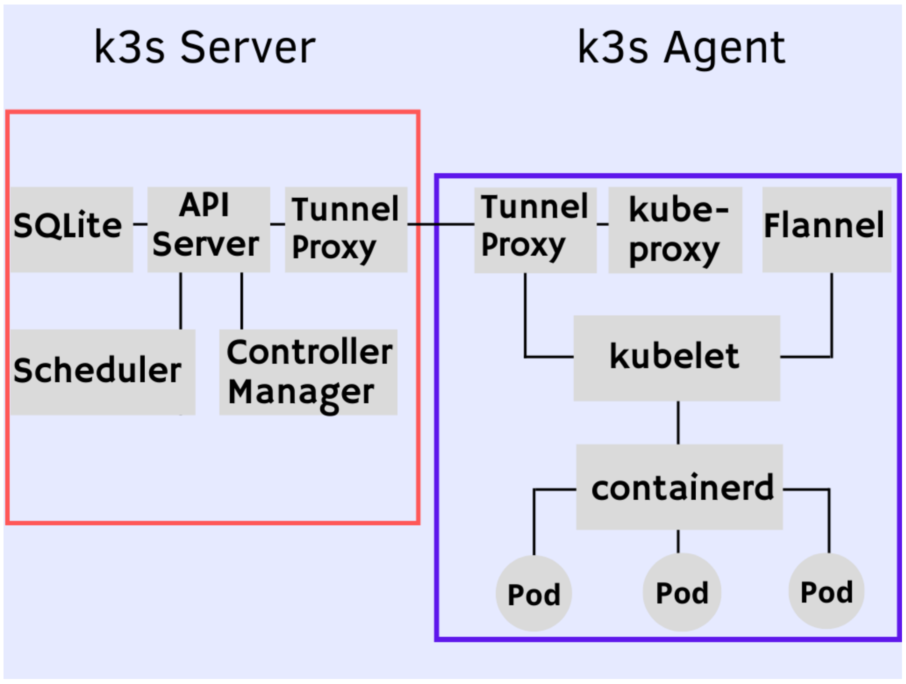


Figure 2.6. k3s Architecture

2.3.6 K3s

K3s is the lightweight distribution of Kubernetes developed by Rancher and it is Cloud Native Computing Foundation (CNCF) certified. It comes packaged in a single small binary, less than 50 MB, which makes it perfect for systems with limited resources, such as IoT devices. It is very easy to be installed and configured, Rancher itself claims that less than 30 seconds are required [22], and it represents a valid alternative in development scenarios in which the complexity of Kubernetes can be put aside in favor of another ultimate goal.

Architecture

From a technological point of view, k3s reflects reliably the Kubernetes architecture described in section 2.3.1, as shown in figure 2.6. For simplicity, the image refers to a single-server setup with an embedded SQLite database. A master node is defined as a machine running the **k3s server** command, while a worker node is defined as a machine running the **k3s agent** command [23]. The default container runtime used by k3s is containerd, while the default CNI is Flannel.

Chapter 3

Trusted Computing, TPM 2.0 and RA

Due to the incredible increase of cyberattacks, trying to protect the software by exploiting the software itself was not enough anymore. The idea behind the Trusted Computing technology was born having in mind that some hardware point of trust was needed since hardware attacks require physical presence and in this way, a better defense would be achieved. For this reason, the Trusted Platform Module was standardized and adopted, allowing the remote attestation process which is fundamental to evaluating a platform state. A platform cannot provide and claim its state, because if it has been tampered with, then any self-analysis will fail. Rather, it can *report it* to an external entity which is commonly called *verifier*.

This chapter focuses on the Trusted Computing technology, the architecture of the TPM 2.0 chip, and the remote attestation process.

3.1 Overview

The Trusted Computing technology emerged in the 90' s, at the hands of the Trusted Computing Group (TCG), to overcome the necessity to establish trust in a device. Being a trusted platform means behaving exactly in the expected way when any operation is performed on the platform itself, or, in other words, the detected behavior needs to be verified against the expected one. This can be achieved by equipping the platform with at least three basic fundamental features [24]:

- the *protected capabilities*, which are a series of operations that allow accessing the so-called *shielded locations*, places in memory such as hardware registers, where is possible to safely operate on data. The protected capabilities are the only commands that possess this permission over the platform.
- the *integrity measurements*, which is the procedure from which the measures of the platform are retrieved and stored in the shielded locations. A measure is obtained by applying a cryptographic hash function to something meaningful to get the integrity status, so it is basically a digest.

- the *integrity reporting*, which is the process that consents the remote attestation of the platform. Through integrity reports, which are exchanged securely between the platform and a remote entity, it is possible to get evidence of the platform's trustworthiness.

According to the TCG specifications and as stated in paragraph 3.2, each one of these features is associated with the correspondent Root of Trust, which must be trusted by definition because their misbehavior might be undetected.

3.2 Roots of Trust

The specific purpose for having Root of Trust components is to get a chain of trust. In fact, in the integrity measurements process, everything is based on a simple assumption: each component has to measure the next one, before execution. This *transitive trust* is giving birth to a chain, and, since the starting point of the chain must be trusted by default, RoTs are needed to guarantee reliance.

According to TCG guidelines, as a minimum, a Trusted Platform should have one root of trust for each fundamental feature of the platform:

- Protected capabilities associated with the Root of Trust for Storage (RTS)
- Integrity Measurements associated with the Root of Trust for Measurements (RTM)
- Integrity Reporting associated with the Root of Trust for Reporting (RTR)

TCG provides also a hardware implementation of the RTS and RTR through the TPM 2.0 chip, a cheap component that is capable among other things of storing and reporting the measurements, as explained in section 3.3. The RTM cannot be implemented through the TPM alone, but it can be used together with other technologies to supply this lack.

3.2.1 Root of trust for Storage

The Root of Trust for Storage is the computing engine responsible to hold in a secure way both the integrity measurements and the keys used by the Trusted Platform to perform cryptographic operations, so it is implementing the concept of shielded locations. A shielded location can contain either sensitive data, such as the private part of a key pair, or non-sensitive data which does not require protection from disclosure, such as integrity digests. Of course, only the non-sensitive information can be reported through the RTR.

3.2.2 Root of trust for Measurements

The Root of trust for Measurements is the computing engine responsible to make integrity measurements and send them to the RTS. Typically the RTM is initiated by the *Core Root of Trust for Measurements* (CRTM) at the boot phase, which guarantees the measured boot, a procedure to establish the integrity of the system when it is switched on. The integrity measurements can concern either program code or embedded data, on which a digest is calculated through a cryptographic hash function.

3.2.3 Root of trust for Reporting

The Root of Trust for Reporting is the computing engine responsible for securely reporting the data maintained within the RTS and attesting its legitimacy to external entities. This procedure is performed by making *integrity reports*, which are digests calculated over some shielded locations and digitally signed to prove the platform identity.

3.2.4 Trusted Platform Building Blocks

Roots of Trust are composed also of parts that do not have shielded locations or protected capabilities, known as Trusted Building Blocks (TBB) [24]. TBB includes platform specific-functions, such as RTM initialization or the connection between the TPM and the motherboard. Roots of trust and TBB provide together the minimum *trust boundary*, which is the collection of trusted components that a platform has at a given point in time. The trust boundary is extended every time a new component is added to the chain of trust, so every time a component is first measured and then executed. As an example, in figure 3.1 can be seen the extension of the trust boundary at system boot with three additional components. The CRTM (which is part of the TBB) together with the Roots of Trust starts the measured boot of the platform in this way: first, it measures the Operating System Loader code, which then is executed. Now, the OS Loader is part of the trust boundary and has to measure the next component before execution, which is the Operating System code. Once the Operating System code has been measured it can be executed, becomes a new trusted component, and repeats the process with the Application code.

3.3 TPM 2.0

The Trusted Platform Module chip is the core hardware component of the Trusted Computing technology, and it could be either added within the CPU or embedded in the device's motherboard. Nowadays the latter is the common case since most devices that have been shipped in the last years are directly equipped with the chip. Originally the TPM was presented in version 1.2, which today has been superseded by the newer 2.0, which offers more functionalities. For this reason, and since the thesis work relies upon it, the following introduction is related to its latest version.

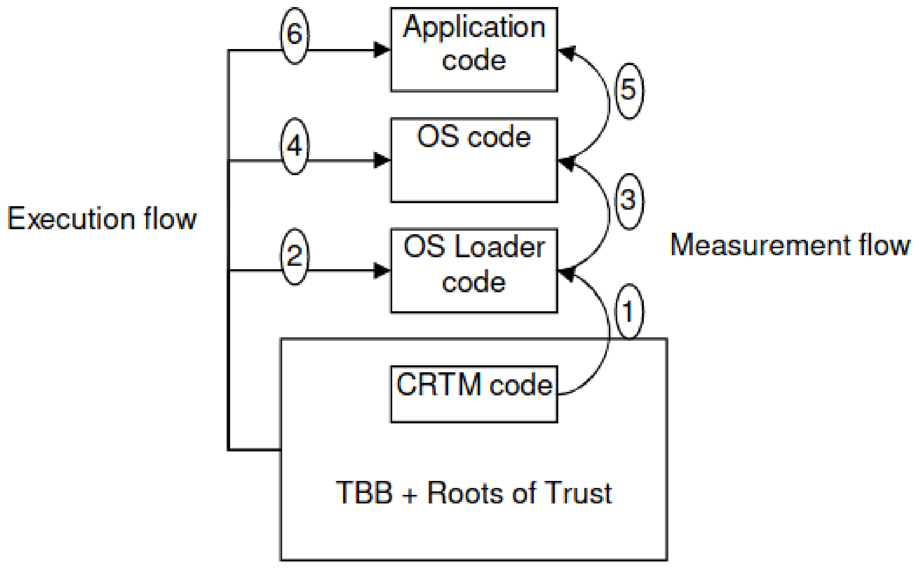


Figure 3.1. Transitive Trust applied to system boot (Source: [24])

3.3.1 Features

The main TPM features can be summarized as follow:

- It can be used as a cryptographic co-processor since it contains a hardware random number generator and it can securely generate encryption keys, offering **platform data protection**.
- It can be exploited as an implementation of the RTS and RTR to consent the **remote attestation** process, explained in detail in paragraph 3.4.
- It can perform **data binding**, a feature more powerful than pure encryption: the data bound to the TPM can be decrypted only by the TPM itself since the used RSA key is unique and it descends from a storage key.
- It can perform **data sealing**. Again, it is a feature more powerful than pure encryption, because the data is encrypted with a key that is stored only within the TPM, but additionally, it is also specified the state in which the system must be to *unseal* (e.i. decrypt) the data.
- Each TPM has a unique and secret Endorsement Key (EK) that was created when the chip was manufactured, which together with the X.509 certificate represents its identity. This feature allows the support to **hardware device authentication**.

3.3.2 Architecture

From an architectural point of view, the TPM's main components can be seen in figure 3.2:

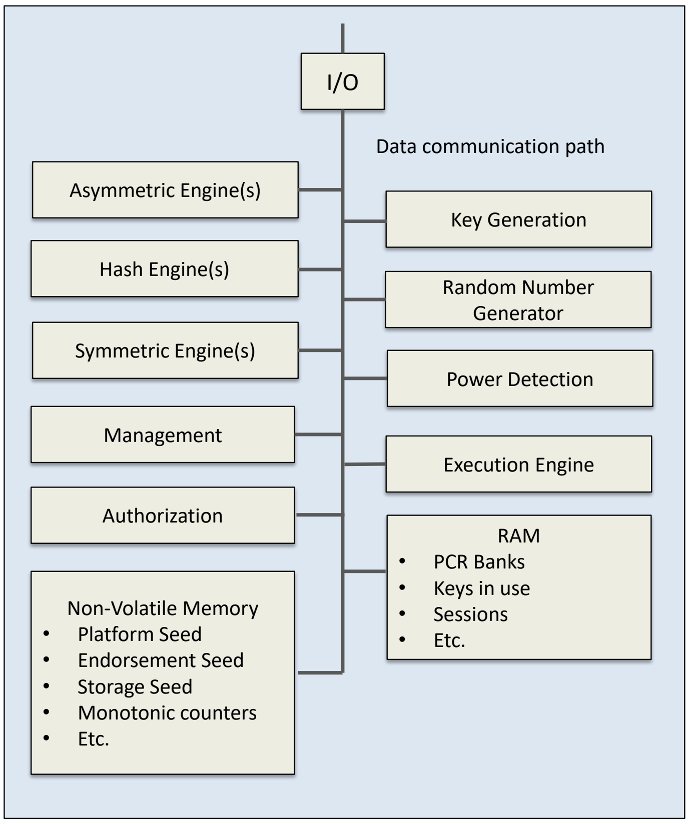


Figure 3.2. TPM 2.0 Architecture. Source: [25]

- The **I/O buffer** consents communication between the host system and the TPM chip: the host system sends command data in the buffer, while the TPM puts the corresponding response.
- As the names suggest, the **asymmetric engine**, **hash engine**, **symmetric engine**, **key generation** and **Random Number Generator** components, are all related to the cryptographic functions which can be called either through the **Execution Engine** or the **Authorization** module.
- The **Power Detection** module is responsible to handle the power state of the TPM that can be ON or OFF. When the platform power state changes this module must be notified to react accordingly.
- The **Random Access Memory** contains the TPM temporary data and a portion of the I/O buffer. This data includes Platform Configuration Register (PCR) banks explained in section 3.3.3, keys and data loaded from external memory, and authorization sessions. When the power state of the TPM is turned OFF, RAM data may be lost depending on the implementation.
- In the **non-volatile memory** is included the information related to TPM persistent state. Data within non-volatile memory can concern either private data meant to remain secret (keys, seeds, and so on) or data that a caller can read, such as monotonic counters. The implementation exploits an index-based data structure, that allows discrimination between TPM's indexes (to

access locations related to TPM specifications) and user-defined indexes (to access locations defined by a user).

- When a TPM command needs access to shielded locations that require authorization, the **Authorization** module is invoked twice, before and after the command execution. Before the execution is checked if the proper authorization is provided, while after is generated a response.

3.3.3 Platform Configuration Registers

Platform configuration Registers are a set of hardware registers that implement shielded locations. They are involved in the process of storing and validating the platform measurements made by the RTM, so they are an integral part of the RTS and RTR implementation. According to the TPM 2.0 specification at least 24 registers must be present, each one used to store measurements of specific modules. As an example, in figure 3.3 can be seen the PCR Mapping of some modules. The

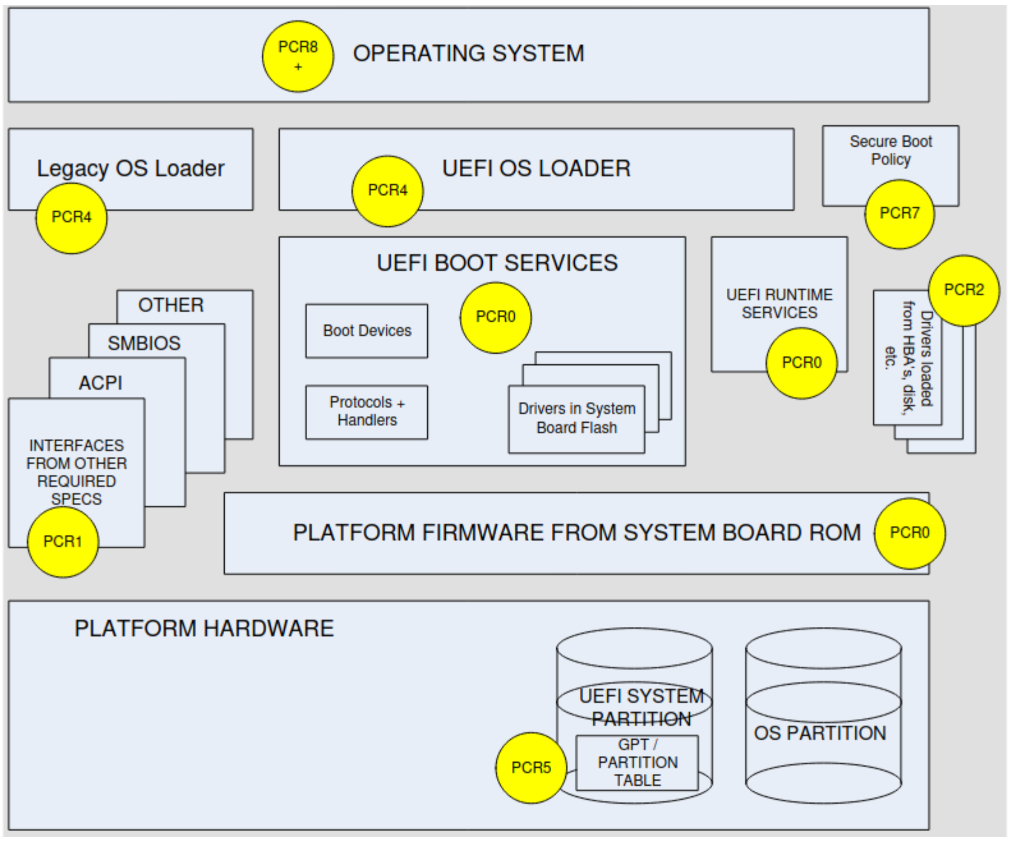


Figure 3.3. PCR Mapping of UEFI Components (Source: [26])

value stored in a PCR can be modified only through two operations, reset and extend. The reset operation brings all the PCRs to their default value, usually all-zeros, and it happens any time the platform is switched on. The extend operation is used to update their content following this procedure:

$$PCR_{new} = Hash(PCR_{old}||Measure) \tag{3.1}$$

The updated value of the PCR is obtained by performing a secure hash function over its old value concatenated with the new measure. The extend operation consents to store a big number of measurements in a single PCR, and since it is involved a hash function in the computation, it is irreversible, a fundamental feature for the evaluation of the platform integrity. Starting from the current value of a PCR, if the list of measurements that have been brought to that specific digest is not known it is impossible to establish it, and it is really hard to obtain the same digest performing the extend on a different list.

PCRs are grouped within a *bank*: a bank is the set of PCRs that are extended with the same hash algorithm. For this reason, the 24 PCRs mentioned above are present in all the enabled banks. The TPM 2.0 specification expects at least the presence of the SHA-1 and SHA-256 banks.

3.4 Remote Attestation

The Remote Attestation (RA) is the process through which the state of a platform called *prover*, is evaluated by a remote entity called *verifier*. As specified by TCG, to be evaluated the prover has to be equipped with a hardware technology that acts as the RTS and RTR, such as the TPM chip, and it has to provide an RTM.

3.4.1 Execution Flow

The logical flow of the process can be summarised as follow:

1. The verifier contacts the prover by sending a Challenge Request together with a nonce. A nonce is *"a number used only once"* to ensure freshness and avoid replay attacks. In the requests are specified the shielded locations values that the prover has to send back.

2. The prover asks the RTR to retrieve the shielded location values from the RTS. This can be done only by exploiting the protected capabilities functions. Then, is generated a response, which will contain also the nonce. The response will be signed with the device's key to ensure hardware device authentication. If the TPM is used as anchor technology, this response is called *quote* and it contains the PCR values specified in the Challenge Request, and it is signed with the private part of the TPM's Attestation Identity Key (AIK). AIKs are keys generated by the TPM starting from its EK.

3. If the prover has not just been asked to provide evidence of its trusted status at the boot phase, the RTM has to be further involved to provide a *measurement log*, before the generation of the *integrity report* from the RTR. This happens because measurements performed after the boot phase leads to unpredictable PCR values, due to the dynamic nature of the operations in a system. This concept will be explained in detail in chapter 4 with the introduction of the Linux Integrity Measurement Architecture module.

4. The RTR generates the integrity report, which will contain the shielded locations, the nonce, and, if specified in the request, the measurement log, and it sends it to the verifier.

5. The verifier first checks the authenticity of the sign and the freshness of the report. If it has to check only the trusted status at the boot phase, it will check the value of the involved shielded locations against *golden values*, which are the expected ones to consider the prover to be trusted. Instead, if a measurement log was asked, the verifier has also to check the integrity of the log itself and then verify that the reported measurements are valid and lead to the provided value calculated through the extend operation.

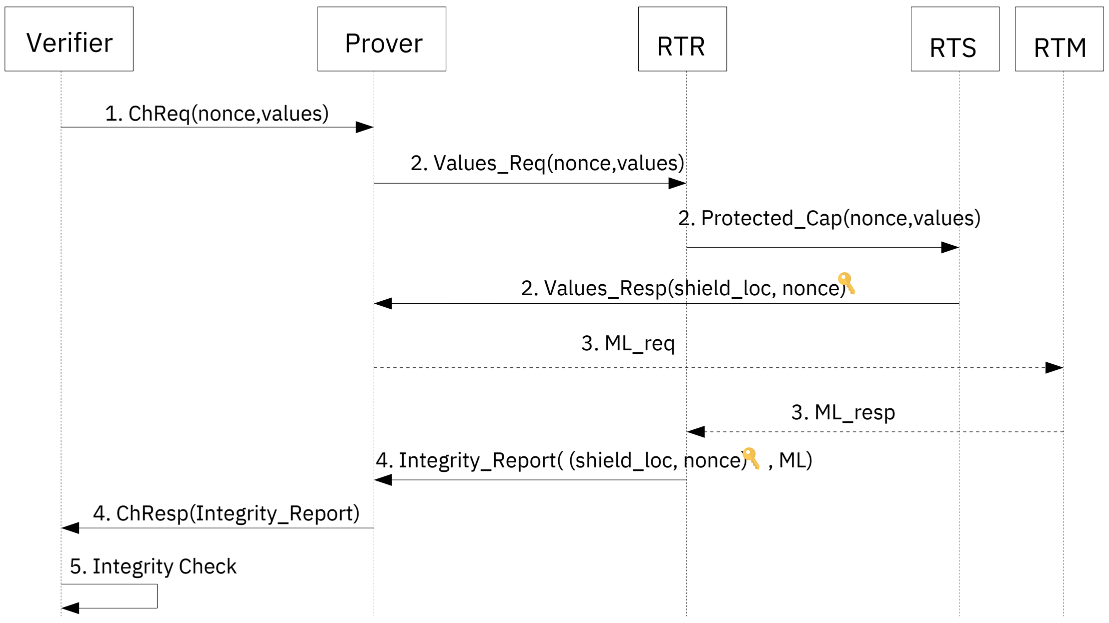

Verifier
Prover
RTR
RTS
RTM
1. ChReq(nonce,values)
2. Values_Req(nonce,values)
2. Protected_Cap(nonce,values)
2. Values_Resp(shield_loc, nonce)
3. ML_req
3. ML_resp
4. Integrity_Report((shield_loc, nonce) , ML)
4. ChResp(Integrity_Report)
5. Integrity Check

Figure 3.4. Remote Attestation Schema

The process is continuously repeated to detect as soon as possible any kind of anomaly. For instance, for the most critical systems, an integrity report may be sent every one or two seconds.

3.4.2 Architectures

Device Identifier Composition Engine

From an architectural point of view, the TCG proposed a layered RA solution named Device Identifier Composition Engine (DICE). DICE splits the system into layers: when the latter is booted, the first layer to be executed is the one containing an *Unique Device Secret* (UDS). RoTs are based on the UDS, which is the starting point of the layered attestation. The current layer computes a secret and attests the next one by signing the *evidence* of its trustworthiness. One evidence is a message containing assertions from the prover [27], so it is the response that is sent when a verifier requests a challenge. DICE is mostly targeting IoT devices, in which an anchor technology such as the TPM may be unfeasible due to hardware requirement limitations.

Remote Attestation Procedures

As an alternative, an accurate collection of architecture workflows can be found in the Remote Attestation Procedures (RATS) Archive [28], which is maintained by the Internet Engineering Task Force (IETF). Specifically, it consists of a series of *draft* documents, meaning that they are temporary. Their validity is extended to a maximum of six months due to their state-of-the-art nature and continuous updating, but they still represent a valid source to consult, present, and exchange mechanisms to perform RA.

3.4.3 Use Cases

The RA process consents to establish the integrity of a system, hence to establish its trustworthiness for a variety of use cases[29]:

- *Authentication*: getting evidence that the underlying system and its authentication mechanisms have not been tampered with, brings to make trust authentication decisions by knowing that the correct identities are used.

- *Release of sensitive information*: when there is the need to exchange sensitive information with a remote entity, by implying RA the risk of broken confidentiality damage is greatly reduced.

- *Network Access control*: adding integrity information about the system that provides some kind of credentials can be useful in taking an authorization decision. This can concern multiple scenarios, from evaluating if a system can join a closed network to a simple web request.

- *Assessing Network resources*: as a provider can be concerned about taking an authorization decision, a client of the service can be concerned about having evidence of the provider's worthiness as well. This permits the detection of a compromised server or any other malicious network device.

- *Remote host monitoring*: monitoring systems aims to report data to be analyzed to a remote server. Coupling the data with integrity evidence brings better assessments and responses.

- *System assessments*: being able to demonstrate the integrity of a system, lead to proof that the system is worthy of the goal it is being used for. This is really important, especially in Cloud Environments, where the runtime integrity data shows that neither the cloud provider nor any other actor, has access to the system.

3.4.4 Trusted Execution Environment

The TPM 2.0 chip was born as a foundation element for the Trusted Computing technology, as stated in 3.3. However, alternative anchor technologies can be used to consent the Remote Attestation process, such as the Trusted Execution Environment (TEE). TEE is a tamper-resistant processing environment that runs on a *separation kernel* [30]. The concept of separation kernel was introduced to emulate a distributed node, where multiple systems coexist in the same platform. Basically, a separation kernel divides its regions guaranteeing high isolation between them and allowing communication only via a protected and monitored interface. This consents to get a specific area where arbitrary code can be executed in a secure and authentic way, collaborating with the RoTs established by the TC. Since TEE is able to provide the integrity system at runtime, it can be exploited to perform the RA process. TEE is a different technology with respect to the TPM because it is not a physically separated chip: it is an area of the processor.

Chapter 4

Integrity Verification of Distributed Nodes: state of the art

4.1 Overview

The Cloud Computing environment is composed of clusters of distributed nodes, each one executing applications within containers, exploiting the lightweight virtualization. The Remote Attestation process explained in chapter 3 has been consolidated and it works well for physical nodes. However, there is still more than one open challenge related to the containerized scenario. This chapter aims to present the Integrity Measurement Architecture Module as RTM and the Keylime attestation framework, to expose the problem of the remote attestation of a distributed node, and some practical solutions which have been proposed to address the container attestation issue, which represent the starting point of the thesis work.

In the following sections, if not differently specified, the TPM 2.0 chip is assumed to be used as anchor technology in the RA process.

4.2 Integrity Measurement Architecture

To verify the trusted status of a platform, the whole measurement process has to start at the very beginning, specifically when the platform is switched on, as stated in 3.2.2. Checking that the BIOS and the operating system have been loaded correctly is undeniably important because it represents the foundation on which the platform will perform any other computation. However, in this scenario there is a crucial assumption to take in mind: all the components needed to boot the system are called in a specific and predefined order. This results in knowing a priori two things:

- the precise order of the components in the chain of trust,

- the precise values that the involved PCRs should have in the end.

In this specific case, we speak of **Static Root of trust for Measurements**, where the measurement process is deterministic. If performing the extend operation over the chain, are obtained the expected PCRs' values, then the platform has not been tampered with.

Once the system is booted and it is in a trusted state, the RTM is responsible to perform a measure anytime an *event* occurs, all the way up into the application layer. An event is any kind of meaningful operation that is happening on the platform (execution of scripts, delete, create or update a file, and so on). Of course, the dynamic nature of the events does not consent to establishing an order as in the boot phase. For this reason, in this case, is needed a **Dynamic Root of Trust for Measurements**, and the Linux Integrity Measurements Architecture (IMA) module is providing one implementation.

4.2.1 Introduction

The IMA module is one Root of Trust for Measurement implementation, both static and dynamic, and it is part of the Linux kernel since 2009. To keep track of the execution order of the events, the platform requires a Measurement Log (ML) file. Any time a new event is detected, the IMA module performs two operations:

- it stores the measure of the event inside the ML, according to a specific template;
- it extends the new measure to a PCR specifically intended to provide application support [26]. Usually, this PCR corresponds to the number 10, and it is also called IMA PCR. IMA PCR protects the integrity of the ML.

In this way, during the RA when a verifier challenges a prover at runtime, it gets the ML and the IMA PCR, so that the extend operation is performed over the content of the ML, entry by entry, and the result can be compared to the IMA PCR value. If they match, then the ML has not been tampered with, and each entry can be individually analyzed. The system will be considered in a trusted status if each measure in the ML, being compared against a list of acceptable values, has brought the system to an expected state. The involvement of a nonce during the RA ensures the ML is fresh, while the ML integrity ensures its completeness and immutability.

4.2.2 Design

The IMA module has been developed with three collaborating components which together consent the RA process: [31]:

- The **Measurement Mechanism** that is the main component in charge to establish what files to measure in the prover, when performing the measures, and maintaining the ML by filling and protecting it. All the latter operations are performed according to a policy that can be customized. The Measurement Mechanism stores the ML as an ordered list in the kernel and performs the related extension to the TPM. The first element of the ML is always the boot aggregate, guaranteeing the Static Root Of Trust functionality.

- The **Integrity Validation Mechanism** that is the component responsible to compare each measure in the ML against a list of acceptable values. An acceptable value is an expected value for that measure, meaning that it is considered to be a trusted digest for that specific event. All the acceptable values are grouped within a *whitelist* for that event. This component verifies also the integrity and freshness of the ML, so it consents to validate the trustworthiness of the prover system.

- The **Integrity Challenge Mechanism** is the component that allows a verifier to challenge a prover when the RA starts. Through the Integrity Challenge Mechanism, the verifier retrieves the ML together with the TPM signed integrity report.

4.2.3 IMA Templates

The ML is filled by the Measurement Mechanism using a specific template. Each template establishes the entry fields which characterized the event that generates the measure. Specifically, the IMA Template Management Mechanism has been introduced to simplify the template management and decouple it from the rest of the code. The Template Management Mechanism defines two data structures: a template descriptor, that establish what kind of event information should be in the ML, and a template field to generate and show data of a certain type [32]. In other words, a template descriptor has a name and a format string through which are identified the involved template fields, separated by the — character. Each template field and each template descriptor can be consulted through the documentation.

For instance, the default template descriptor used by IMA is called ima-ng, and it is composed of the following format string: d-ng—n-ng. The template field d-ng is referring to the event digest calculated with an arbitrary hash function, while the template field n-ng is referring to the event name without size limitations [32]. As shown in table 4.1, a ML line created using the ima-ng template will have five template fields:

PCR	**template-hash**	**template-name**	**file-hash**	**file-path**
10	dc3b9[...]5f88b	ima-ng	sha1:1ba[...]0ef49	/usr/bin/cp

Table 4.1. A Measurement Log entry created with the ima-ng template

- The field PCR refers to the PCR number used to extend the current entry.
- The field template-hash contains the digest calculated over the template fields of the current entry. This digest is extended in the PCR specified in the first field. The SHA-1 algorithm is used.
- The field template-name specifies the template descriptor used for the entry.
- The field file-hash contains the digest of the file specified in the last field: it is the actual *measure* of the event. The SHA-1 algorithm is used by default.
- The field file-path contains the file pathname.

The first three template fields are mandatory. With the introduction of the Template Management Mechanism it is possible to define new template descriptors, either customizing the format string with the existing template fields or eventually adding new ones.

4.2.4 ML Integrity Verification and Entries Validation

The IMA Integrity Challenge Mechanism starts the RA: the verifier challenges a prover and retrieves the integrity report containing the signed TPM quote and the ML. The verifier first checks the validity of the sign and then, through the IMA Validation Mechanism, it has to perform two macro operations in order to establish if the system is in a trusted state or not.
The first macro operation is the integrity check of the ML itself, so that if it has been tampered with, it is not complete, or it is not fresh, this will be immediately detected and the system considered untrusted. As shown in figure 4.1, this process starts by taking the template-hash field of the first ML entry, which is related to the boot aggregate. The template hash is concatenated with the digest of all zeros, and a digest over the result is calculated. From this point on, the extend operation is performed entry by entry, until the ML finishes and the result can be compared with the IMA aggregate, which usually is stored in PCR 10.

Once the integrity of the ML has been verified, the second macro operation concerns the validation of each entry. Each entry corresponds to an event, and each event has a whitelist associated with it. The whitelist contains the file-path and the list of file-hashes considered trusted for that event. If the file-path associated with the event is not present in the whitelist means that an unrecognized program has been executed on the prover and it is considered untrusted. If the file-path is present in the whitelist but the file-hash of the entry is not present among the acceptable values it is possible that there has been an attack and the platform is considered untrusted as well. When the system is found untrusted, it is the job of the verifier to take the corresponding actions according to a policy. If the file-path and the file-hash of each ML entry are present in the whitelist, then the system is successfully evaluated as trusted.

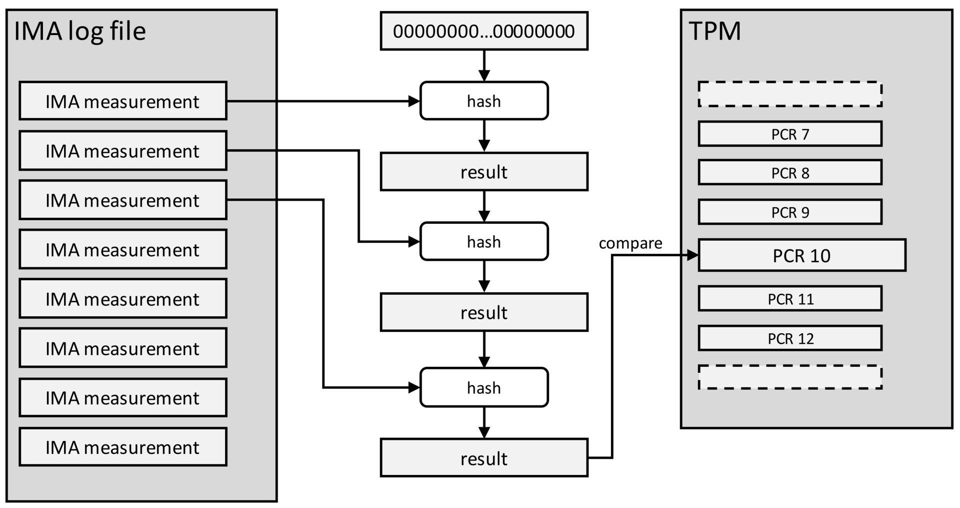


Figure 4.1. IMA ML Integrity Verification (Source: [33])

4.3 Keylime Framework

Nowadays several remote attestation frameworks have been developed and adopted. In this section will be presented Keylime, an open-source and CNCF-certified tool originally developed by the Massachusetts Institute of Technology (MIT). Keylime framework consents the RA in distributed environments, offering a simple way to adopt the trusted computing in the cloud infrastructure. It uses the TPM 2.0 chip as RTR and RTS, while it uses the IMA module as RTM. Keylime is based on the assumptions that an attacker is not able to gain physical access on the platforms and that the cloud provider is *semi-trusted*. Being semi-trusted means that the cloud provider is assumed to put in place some kind of control systems and policies to reduce the consequences of an attack, but he could still have access to tenants' confidential data which could be dangerous in case of a malicious administrator[33].

From an architectural point of view, in the following subsections will be presented the major components involved in the RA process.

Keylime Agent

In the Keylime architecture, the prover system (i.e. the targeting of the RA) is named *"attester"*. Each attester machine runs a Keylime Agent, responsible to send the integrity reports needed to evaluate the integrity state. The Keylime Agent is identified through *Universally Unique Identifiers (UUIDs)*, and when it is kicked off the *Registration Protocol* starts.

Keylime Registrar

The Keylime Registrar is in charge to perform the *Registration Protocol* for the cloud nodes which are running the Keylime Agent, as explained in section 4.3.1. It retrieves and stores the TPM public AIKs, which are needed to perform the quote validation. It sends the TPM credentials with a server authenticated TLS channel to the Verifier component.

Keylime Tenant

The Keylime Tenant represents the human or the organization that is starting and using the services within one or more cloud nodes. It communicates with the Keylime Agent of the node either through a Command Line Interface or REST APIs. After the end of the *Registration Protocol* of the node, the *Three Party Bootstrap Key Derivation Protocol* starts, which involves the Agent, the Tenant, and the Verifier. The latter aims to offer *secure payloads*, used to provide encrypted data to an enrolled node [34]. Basically, the Tenant creates a bootstrap key, which will be cryptographically split into two parts, U and V. The V part is sent to the Verifier, while the U part is sent to the Agent. The Agent will retrieve the V part from the Verifier in a secure way, so it can compose the key. Once the key is established, the tenant sends an *encrypted payload* to the Agent, that specifies the services he wants to deploy in a secure way. The Keylime Tenant communicates also with the Keylime Verifier, to provide the information needed to attest the cloud nodes that are running its services, such as the whitelists.

Keylime Verifier

The Keylime verifier is the component in charge to evaluate the integrity status of each Keylime Agent (i.e. cloud node). It retrieves the public AIKs from the Keylime Registrar to validate the quotes present in the integrity report.

Software Certification Authority

The Keylime Software Certification Authority (CA) has the specific purpose to avoid having each cloud service TC aware, by propagating the trustworthiness from the hardware RoTs to higher level services.

Keylime Revocation Service

The Keylime Revocation Service collaborates with the Keylime Software CA by sending a *revocation event* any time a cloud node is detected untrusted. Since a cloud node is providing some kind of service, also the nodes that are registered for the service should be notified of the revocation event: this is done by exploiting a Certificate Revocation List (CRL) maintained by the software CA.

4.3.1 Registration Protocol

The Registration Protocol involves the Attester Machine that is running the Keylime Agent and the Keylime Registrar, as shown in figure 4.2. The steps are the following:

1. The Keylime Agent sends to the Registrar its Universal Unique Identifier together with the public parts of the Endorsement and Attestation Identity RSA Keys. In addition, is sent also the EK X509 certificate. The Endorsement Key is permanent and certifies a valid TPM, while the Attestation Identity Key is the one used to sign the quotes. The AIK is not fixed to the hardware, meaning that it changes at TPM reset.

2. The Keylime Registrar checks that the EK certificate is signed by an actual TPM manufacturer and then challenges the Agent to decrypt with the private part of the EK key an ephemeral key Ke.

3. When the Keylime Agent solves the challenge, it proves to possess the private counterpart of the Endorsement Key, so it can send back to the Registrar the keyed-hash message authentication code (HMAC) calculated with the UUID and the Ke.

Once this procedure has terminated, the Registrar store the AIK of the Agent.

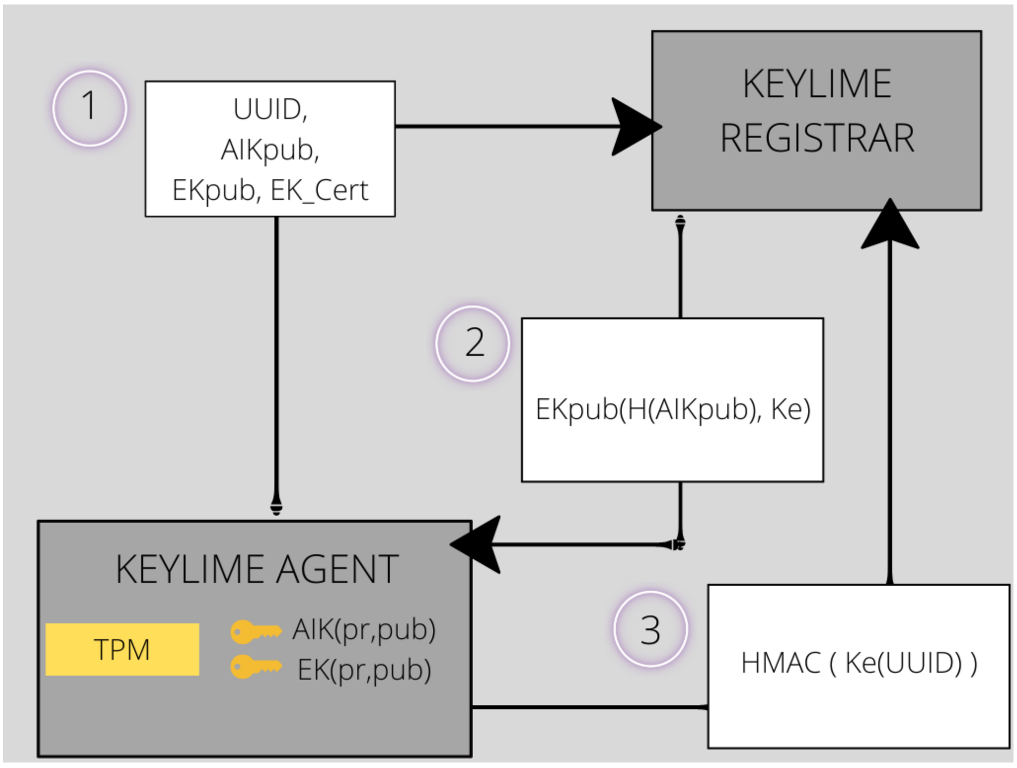


Figure 4.2. Keylime Registration Protocol

4.3.2 Continuous Remote Attestation

After the delivery and the decryption of the *encrypted payload*, the services start on the node and the remote attestation process can begin. As shown in figure 4.3, the Keylime Verifier starts to continuously poll the Agent to retrieve the integrity reports, ensuring that the system stays trusted over time. For each integrity report, the Verifier will check:

- The signature of the quote, validating it with the TPM AIK retrieved from the registrar.
- The freshness of the quote, checking it with the provided nonce.
- The presence of all the PCRs specified in the PCR mask.
- The integrity of the ML, by computing the extend operation, and comparing the result with PCR 10.
- The validity of each measure, by comparing with the whitelist.

By default, the time interval between one attestation and the other is two seconds, but this can be set differently. If the system is detected untrusted, the Keylime Revocation Service is triggered by the Verifier and the Software CA component updates its CRL. Each node registered to the services running on the untrusted one is free to take action accordingly to the revocation event.

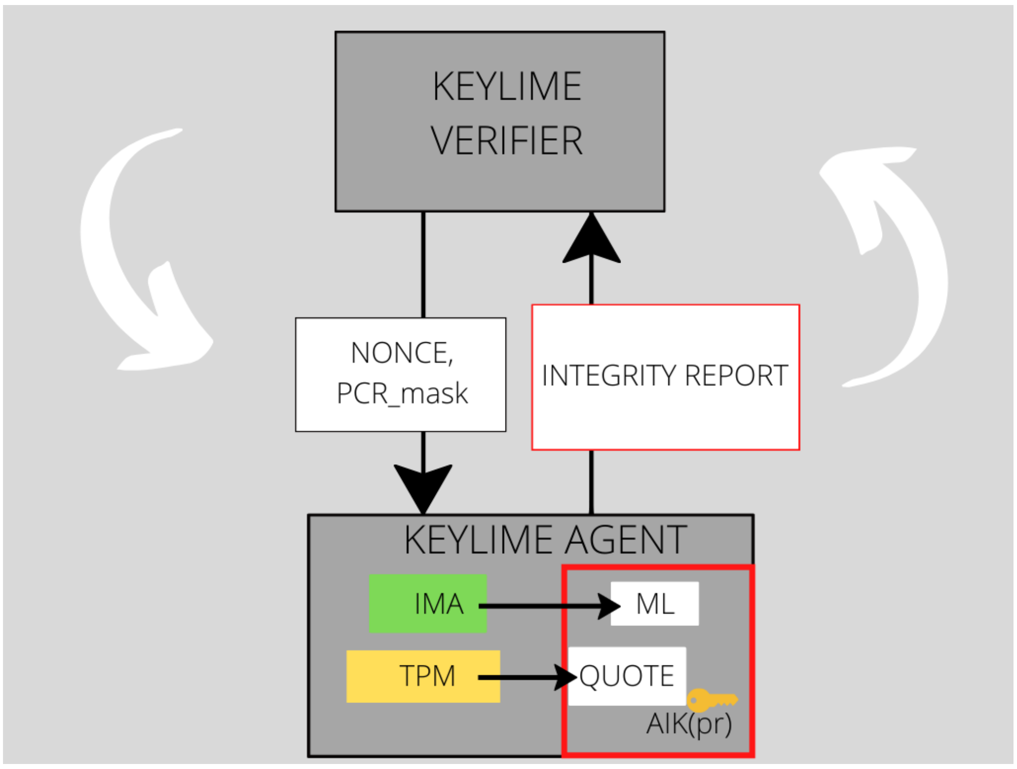


Figure 4.3. Keylime Continuous Attestation

4.4 Container Attestation

The RA process based on Trusted Computing Technology works well to validate the integrity of physical systems. However, the Linux IMA module is built to take and store the measures in a unique ML. If the physical system is a cloud node, it will run services of more than one tenant within containers. The IMA module is not allowing discrimination between ML entries related to the underlying system and the containers, bringing a privacy issue. Tenants should be aware only about the integrity status of *their* services, following the *need-to-know* and *least privilege* security principles. Furthermore, if a specific container measure is found to be untrusted, this will result in considering the whole prover system untrusted as well. A better strategy could consider that it is sufficient to terminate only the untrusted containers, avoiding the system restarting and improving the efficiency of the offered infrastructure. Moreover, according to the Application Container Security Guide of the National Institute of Standards and Technology, the attack surface of container technologies is made large by the variety of core components involved: images, registries, orchestrators, containers, and the host operating system itself [4]. A successful attack against any container running on the system can be spread toward other tenants' containers, or worse, to the host system itself, making integrity verification a fundamental process.

In the following paragraphs will be presented three solutions developed to overcome this issue, underlying their advantages and disadvantages, as a prelude to the solution implemented during the thesis work.

4.4.1 Container-IMA

Container-IMA is a solution for the container attestation developed by Peking University. The main goal of this solution was to overcome the privacy issue related to the multi-tenancy nature of a cloud node. When following the traditional RA pattern for physical nodes, the verifier gets the whole content of the prover ML, even if it has been asked to validate only the containers of a given tenant. This results in giving benefits to potential attackers since they can explore the vulnerabilities of the prover and begin capable of stealing container information of other tenants [35]. For this reason, Container-IMA proposes a modified version of the Linux IMA module, to consent division of the ML entries according to the specific container that generates them.

New IMA Components

To realize the ML subsets related to a given container, Container-IMA enhances the IMA module by adding three components [35]:

- The *Split Hook* component. Since the lightweight virtualization is based on the namespace feature to isolate the system resources, each container should belong to a different namespace. The Split Hook is in charge to inspect the system call related to the namespace creation, to notifying the kernel of an event for ML partitioning.

- The *Namespace Parser* is the component that retrieves the namespace number of the process that generates an event to be measured. In this way, when an event occurs it is possible to collocate it in different ML partitions according to its namespace number (i.e. container that generates the event).
- The *container-PCR Module* (cPCR) is a kernel data structure that simulates the host TPM PCRs to protect each ML partition. Specifically, each cPCR has three fields: the value obtained by applying the extend operation over the container applications partition, the namespace number, and a secret. The secret is used to prevent a verifier from retrieving other tenants' container data.

Architecture

Container-IMA Architecture works with two mechanisms: the *measurement* and the *attestation*. The measurement mechanism is executed on the prover machine and relies upon the modified IMA module. It involves two procedures, the *Namespace Register Procedure* that splits the ML using the Split Hook and the Namespace Parser, and the *Measurement Procedure* that stores each event in the correct ML partition, using also the auxiliary cPCR for the extend operation. On the other hand, the attestation mechanism consent the RA of a specific container, exploiting a modified Attestation Agent that identifies in the prover only the ML partitions of interest. Specifically, if the challenge sent to the Attestation Agent contains the nonce and only one container to be attested, the integrity report will contain [35]:

- the TPM's quote, containing the value of PCRs 0-7, 11, and 12.
- the old value of PCR 12 and the content of the cPCR XORed with its secret. Since the xor operation is the reverse of itself, this is done to prevent disclosure.
- four involved ML partitions, one for the prover measured boot, one for the container measured boot, one for the container dependencies on the prover machine, and one for the container applications.

Since only the needed ML partitions are sent in the integrity report, the attestation procedure guarantees multi-tenant privacy, and since it requires as well the presence of a signed TPM quote, the trustworthiness can be determined with hardware RoTs base. Once the validity of the sign and the freshness of the TPM's quote is checked, the first step on the verifier machine is to check the integrity status of each ML partition by using the content of physical PCRs, with the extend operation. Specifically, with PCR 0-7 is checked the integrity of the prover measured boot partition, with PCR 11 is checked the integrity of the container measured boot partition, and with PCR 12 is checked the integrity of the cPCR value (and, for this reason, also its old value is sent). Only if the ML partitions related to the container dependencies and applications are authentic, the verifier starts to validate their content, comparing their content against the expected values.

Considerations

Container-IMA solves the problem of tenant privacy in the containerized scenario, reducing the latency of the RA process, and without the need to modify the container architectures. However, some challenges still remain open. First of all, Container-IMA relies upon the TPM 1.2 specification, which nowadays is considered deprecated. Another crucial point is that this solution assumes that the RA of the whole prover system is not needed, while the underlying environment should be always checked as it represents the foundation on which the containers are running. Furthermore, Container-IMA does not support containers that share the same namespace, as it happens in the Kubernetes architecture for containers belonging to the same pod.

4.4.2 Docker Integrity Verification Engine (DIVE)

DIVE is another solution for container attestation, developed by the Polytechnic of Turin. It specifically targets the whole underlying system and the containers launched on it through the Docker container runtime, offering a complete integrity verification. Each node of cloud infrastructure can be analyzed through DIVE, as long as it owns a TPM chip. DIVE solution can replace the compromised containers without stopping the whole system, which makes it usable in real scenarios and improves the efficiency of the RA. This is realized by adding a new IMA template, which adds in the ML entries the field *device identifier*. The device identifier is assigned by the Docker's Device Mapper storage driver, and it is different for each container. This additional field bound an event to be measured to the container (or the host) that generates it, hence allowing the discrimination in the ML.

Architecture

The overall architecture of DIVE includes three main components [36]:

- The **Attester**, that is the node to be attested. It is equipped with a TPM chip, uses IMA as RTM, and runs containers through the Docker engine. The Attester runs also a *RA agent* to handle the RA process.
- The **Verifier** is the component in charge to make decisions about the integrity state of each attester and the containers hosted on it.
- The **Infrastructure Manager** that is the component responsible to keep track of the Universal Unique Identifiers (UUID) of the attesters and containers.

When the tenant of the cloud wants to retrieve integrity information about one of its services, the Infrastructure Manager contacts the Verifier and sends the list of the containers and attesters that are running them. The Verifier communicates with the RA Agents of the involved nodes, retrieving the integrity reports. Since the Verifier possesses the list of the containers, it will be able to check the ML entries related to them through the device identifier field and validate their content against a whitelist. Finally, the Verifier sends an attestation report to the Infrastructure Manager. If the attestation manager finds out that the involved nodes are trusted but some containers are not, then it is in charge to replace them without stopping the service.

Considerations

DIVE is a valid solution for implementing the Docker container attestation process since it relies upon trusted computing techniques. Modifications on the container architectures and the Verifier system are not required, the only ones needed to implement the solution concern the cloud node (i.e. the Attester). Furthermore, the performance impact of this solution is almost negligible, consenting usage in real applications. Unfortunately, also DIVE uses the deprecated TPM 1.2 specification, and it cannot provide protection against in-memory attacks [36]. An additional limitation of this solution is that a previously used device id may be assigned to another container. Specifically, this can happen when a container terminates its life cycle, resulting in the presence of a new container inheriting the ML entries of an old one. Of course, this can eventually bring attestation failures.

4.4.3 Docker Container Attestation

The generic name *"Verification of Software Integrity in Distributed Systems"* refers to the latest solution for the Docker Container Attestation developed by Polytechnic of Turin. This work is based on the identification of the ML entries related to containers. As DIVE, the implementation of the solution has been done through an additional IMA template, but in this case, the TPM 2.0 specification is used. In this case, the field template-hash is computed using the SHA-256 hash algorithm, to exploit the powerful functionalities offered by TPM 2.0. Keylime is the RA framework involved, which has been affected by some modifications needed to make it capable to register containers and understanding the new template.

IMA template

The new IMA template was needed to bind an event to be measured to the container that generates it. Control groups are a Linux kernel feature exploited to organize the processes into hierarchies, to guarantee the resource limitations needed in the lightweight virtualization technique. In the Linux kernel, every process within a container inherits the same control groups of the process that generates the container, which is called *init process*. It was observed that for the init process, the Docker engine automatically assigns the container-full-identifier in various control group names, meaning that processes within a container can be identified by any one of their control group names. For this reason, two additional fields were added: the control group name and the dependencies of the process, following the template descriptor dep—cgn—d-ng—n-ng.

Adding this new template permits a Verifier to bind an ML entry to a container only if:

- it has the control group name field filled with a container-full-identifier,
- it has a container runtime in the dependencies list field,

otherwise, the entry will be considered as belonging to the host system.

Keylime Modifications

The Keylime Modifications concerned the Agent, Verifier, and Tenant components, since the operation involved by the presence of the new IMA template were the registration of the containers at the Verifier and the IR creation and validation. In this solution, the registration of the containers in the Verifier can happen either at the moment of the Agent registration or later and expects to be provided with the container IDs and whitelists. In this way, to be evaluated each container can be treated exactly like the host, discriminating its ML entries and comparing the measures against its whitelist. In this sense, the validation of the integrity report was affected. Finally, the IR creation has been affected to send just a portion of the ML, since the used Keylime version was sending the whole content of the ML at each attestation cycle.

Considerations

This solution permits the RA of individual containers, meaning that whenever a container is found untrusted there is no need to reset the whole platform. Furthermore, the solution takes advantage of the powerful SHA-256 algorithm offered by the TPM 2.0 chip and reduces the latency of the attestation by sending only a portion of the ML. This work represents a valuable starting point to realize other solutions for the container attestation, because can be adapted and integrated with the Kubernetes scenario, in which as stated in 2.3.1 the smallest scheduling unit is not the container.

Chapter 5

Remote attestation of a Kubernetes cluster - Design

This chapter proposes the design of the implemented solution. In the beginning, an overview of the problem statement in the cloud scenario will be exposed, explicitly targeting the Kubernetes orchestrator as motivation for the thesis work carried out. Then, the approach to solving the problems faced will be presented, following a step-by-step approach. The starting point will be the discrimination of each Measurement Log entry between the attester system and pods, followed by the Keylime integration. The description of the high-level architecture of the proposed solution will be provided to underline how the various components communicate with each other and to show how they can be integrated with the Kubernetes ones. Finally, will be exposed the application of the solution to multiple nodes (i.e. to a cluster) together with some final considerations concerning the used approach.

5.1 Problem Statement

The solutions exposed in chapter 4 are not always suitable for every scenario, especially when we consider the current situation. Container-IMA and DIVE are solutions that rely upon TPM 1.2 specification, which nowadays should be replaced by the newer 2.0. The Docker Container Attestation solution works fine and uses the 2.0 specification, but targets the Docker container runtime which is slowly being superseded by newer and OCI-compliant container runtimes, as stated in section 2.2.6. Moreover, since nowadays Kubernetes represents the de-facto standard for the orchestration technology, the need to implement a solution for the remote attestation that targets the *pod* and no longer the *container* emerged. The reason is that Kubernetes does not work with containers, since the smallest scheduling unit is represented by the pod, as stated in section 2.3. Containers within a pod share the same namespace, and the Container-IMA solution does not offer this kind of support. Furthermore, within a pod, it is sufficient to have at least one compromised container to affirm that the pod is untrusted, hence there is no more the necessity to identify the specific container which causes the problem. Simply, whenever one of the pod's containers is performing unexpected executions, the pod is knocked down and rescheduled with a fresh and uncompromised version, without affecting the integrity of the host system. So, the first thing to underline is that to develop an efficient solution the attention has to focus on the pod concept and on the variety of container runtimes that can be used to launch it.

In a cloud environment handled by the Kubernetes orchestrator, there are hundreds of nodes where pods have to be instantiated. Each node is hosting pods for running applications belonging to different tenants. To guarantee the privacy and make possible the remote attestation of each one of them, there should be a bind between the pod itself and the specific tenant who asked for it. In this way, the tenant requiring an assurance proof of the nodes of interest (i.e. the nodes running its applications), could retrieve only the information about its own pods and nobody else's. This kind of information can be retrieved by the orchestrator, with a proper integration in the control plane which knows exactly this kind of bind.

Once established that the privacy issue could be easily superseded, can be observed that another important problem is represented by the ephemeral nature of pods, meaning that they change their identifiers any time they are rescheduled. For instance, the latter situation can happen when a node needs to be restarted, and in that case, not only the identifier changes but also the physical node in which the pod will run, growing the complexity of the problem exponentially. This is representing a migration problem to be solved, because a tenant who possesses the lists of its pod identifiers and nodes in which they are running, at some point in time could find these lists completely changed, losing control of the integrity check on the infrastructure. Of course, the updated lists can be retrieved dynamically from the orchestrator as well, but from the point of view of a node, how can a Verifier know a priori which pods are allowed and which are not, if they are continuously changing their location?

A possible solution to the migration problem is to put a constraint: physical nodes have to stay synchronized, to guarantee simpler management. In this context, being synchronized means having a general list of allowed pods, the ones specified by each tenant of the cloud. In this way, whatever node of the cloud is selected to run the pod, the pod will be considered as allowed (i.e. put within the list of allowed pods) as long as at least one tenant specifies that pod. Of course, a solution like this lacks in terms of optimization but consents to avoid the underlying node being considered untrusted at the moment when a new pod is rescheduled within it. In other words, identifying a pod and solving the migration problem is one of the biggest advantages of a solution like this. However, why this solution lacks in terms of optimization? On one hand, we can consider that the pod list resides in Verifier nodes which in general are fewer than the nodes to be attested, but on the other, each pod is associated with its own specific whitelist of golden values to be used in the verification process. For instance, if a node is running 50 different pods, then the Verifier must have that 50 pods in the list and 50 different whitelists, one for each pod. If we add another node, with 50 additional different pods, the whitelists on the Verifier side pass from 50 to 100, and so on. The number of whitelists grows very quickly and easily, with a huge waste of memory and with the growth of the attack surface, because in each Verifier node are present whitelists of pods that are actually not running.

Then, going more in-depth with the reasoning, how to bind an event to be analyzed to the entity that generates it? For instance, the objective is to treat the host and pods separately, to allow a node to remain trusted even when one of the pods may not be. This means that an event can be verified either against the host whitelist or any whitelist among those of the pods. So, the question is: the event that occurs on the platform, was it triggered by a pod or by the host system? Following the DIVE and Docker Container Attestation approaches, this can be done using a patched version of the kernel with a new template to fill the Measurement Log. However, a solution that assumes the usage of a patched kernel possesses strict requirements and from the management point of view is inconvenient, since has a considerable impact on the cloud infrastructure. In fact, being strongly linked to a specific kernel version brings physical changes to the nodes of the infrastructure, which cannot always be a suitable choice. On one hand, this represents one of the most important drawbacks derived from choosing a new template, but on the other, adding a new template that is also able to identify the specific container runtime that launched that pod consents compliance of the solution with any of them.

Finally, talking about the hash algorithm to be used in the measures of the events, it would be better to have a flexible choice that consents to pass easily from one algorithm to another one, according to the specific device to be attested. This can be done again by intervening on the kernel, with a command line parameter, with the same issues reported for a patched kernel. In other words, it is not that easy to scale the cloud infrastructure, and each proposed solution has its own pros and cons to be evaluated before taking a decision.

5.2 Proposed Solution

The Linux IMA module was born when the Root Of Trust for Measurement and the remote attestation process was targeting almost only physical systems, so it was not developed thinking about splitting the measures into more ML files. Examining a unique ML is enough to establish the trustworthiness of a physical system, but this is not an applicable solution for the cloud environment. The choice of an efficient strategy to overcome this IMA limitation represented the first problem to face during the thesis work. To consent to an untampered cloud node to run properly and without interruptions even if a pod is evaluated untrusted, the ML has to provide in some way a method to establish if each entry has been generated by the prover system or by a pod running within it. The solution proposed by Container-IMA added new components to split the ML file, to get container-specific MLs. However, the latter was not considering the integrity verification of the underlying platform, a fundamental requisite in the cloud scenario. For this reason, as for DIVE and Docker Container Attestation, the choice fell on implementing a new IMA template that consents to affirm with certainty the entity that generates the ML entry, as explained in section 5.3.

A second choice to make was represented by the Remote Attestation framework to be used: since Keylime is an open source framework and it relies upon the TPM

2.0 specification, it was a good fit for the work purpose. Of course, the framework had to be modified to understand the new IMA template and work accordingly in the validation of the ML, as stated in section 5.4.

The last choice to be made regarded the cluster set-up, specifically for the testing phase. Once the new template has been defined, works with the Keylime framework, and the remote attestation works for a single node that is running pods, it was time to check if a Kubernetes Master Node running the Keylime Verifier was able to attest at least two Kubernetes Worker Nodes running the Keylime Agent. For this last purpose, due to the complexity of Kubernetes and to avoid investing too much time, the lightweight k3s version has been chosen to focus the attention on the main objective of the work. The Cluster Set Up will be explained in section 5.6 with the application of the solution to multiple nodes.

In the following sections will be presented the solution development following a step-by-step approach.

5.3 Measurement Log Discrimination

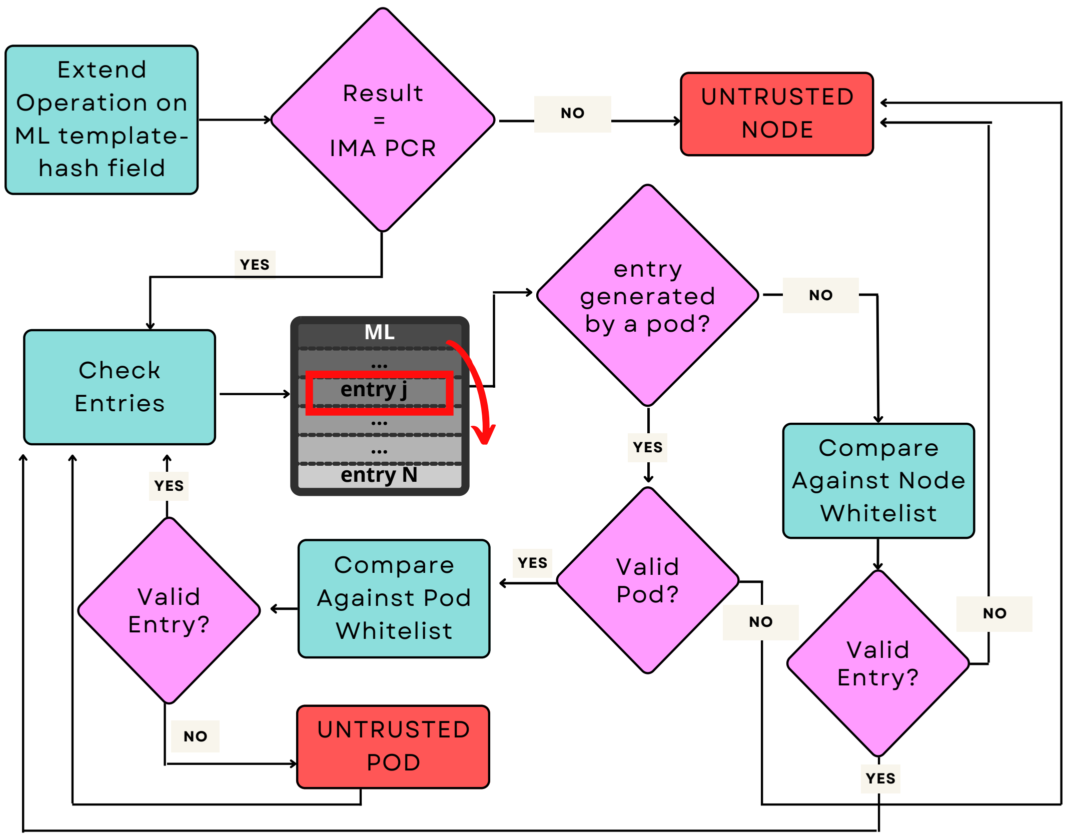
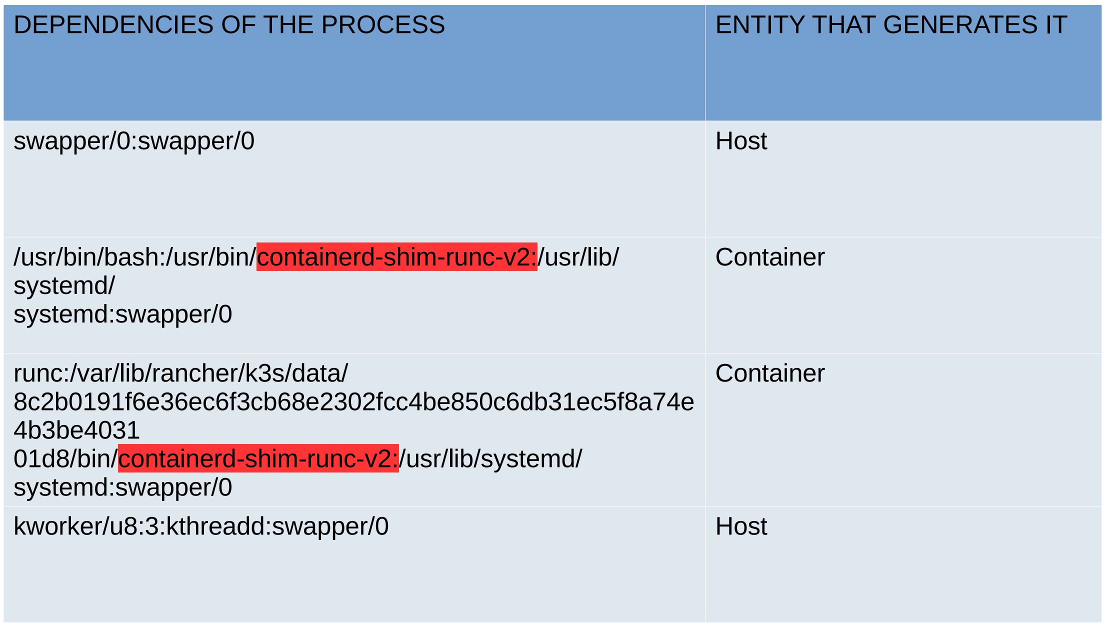


Figure 5.1. Schema Of The Result to be Achieved

The starting point of the work concerned the association of one ML entry to the *entity* that generates it. In this case, the entity may be either the attester system (i.e. the Kubernetes worker node) or any pod running within it, and the default IMA templates do not consent to make such discrimination. In figure 5.1 can be seen the schema with the result to be achieved. To think about a solution and bounding an event to a specific pod, was useful to study and take into account the previously proposed approaches. The ML is filled with a different entry whenever a new operation is performed by a process. A process is a running instance of a generic program: for instance, a container is seen by the physical system exactly as a process. As a result, the first observation made by Docker Attestation developers was that in the dependencies of the process that generates a container-related entry, there should be some hint about the container runtime that launched it. This means that the dependencies of the process that generates the entry are the first IMA template field to take into consideration, to make a preliminary skimming: if a container runtime is present, then it is unlikely an entry that belongs to the host system, as shown in figure 5.2. Then, a docker container can be uniquely identified

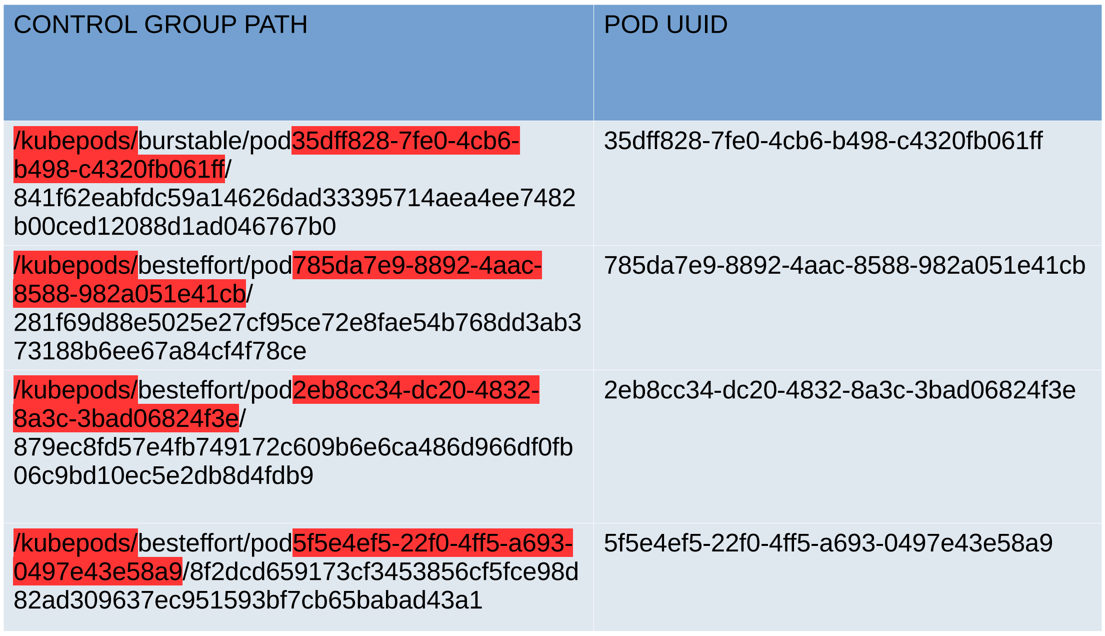


DEPENDENCIES OF THE PROCESS	ENTITY THAT GENERATES IT
swapper/0:swapper/0	Host
/usr/bin/bash:/usr/bin/containerd-shim-runc-v2:/usr/lib/systemd/ systemd:swapper/0	Container
runc:/var/lib/rancher/k3s/data/ 8c2b0191f6e36ec6f3cb68e2302fcc4be850c6db31ec5f8a74e 4b3be4031 01d8/bin/containerd-shim-runc-v2:/usr/lib/systemd/ systemd:swapper/0	Container
kworker/u8:3:kthreadd:swapper/0	Host

Figure 5.2. Depedencies Field

either through its device identifier, if we consider DIVE, or through its container identifier, if we consider Docker Container Attestation. At this point of the reasoning was observed that the latter strategy was the better, since there were no problems related to containers inheriting previously used identifiers, as happened with DIVE and the device identifiers. Moreover, the container identifier is not a Docker-specific characteristic. So, the problem passes to "what is the unique characteristic of a pod, that can be inserted within an entry?". Obviously, the answer was found in the pod identifier. Specifically, in Kubernetes, every object is assigned with a different universally unique identifier (UUID) to distinguish between historical occurrences of similar entities [13].

Once established that the pod UUID was needed, the next problem was how to retrieve and put it in an IMA template field. By examining the control group feature offered by the Linux kernel, was observed that the control group path related to a pod handled by Kubernetes, starts always with the same string, and it contains exactly the pod UUID, as shown in figure 5.3. Exactly as the container processes



CONTROL GROUP PATH	POD UUID
/kubepods/burstable/pod35dff828-7fe0-4cb6-b498-c4320fb061ff/841f62eabfdc59a14626dad33395714aea4ee7482b00ced12088d1ad046767b0	35dff828-7fe0-4cb6-b498-c4320fb061ff
/kubepods/besteffort/pod785da7e9-8892-4aac-8588-982a051e41cb/281f69d88e5025e27cf95ce72e8fae54b768dd3ab373188b6ee67a84cf4f78ce	785da7e9-8892-4aac-8588-982a051e41cb
/kubepods/besteffort/pod2eb8cc34-dc20-4832-8a3c-3bad06824f3e/879ec8fd57e4fb749172c609b6e6ca486d966df0fb06c9bd10ec5e2db8d4fdb9	2eb8cc34-dc20-4832-8a3c-3bad06824f3e
/kubepods/besteffort/pod5f5e4ef5-22f0-4ff5-a693-0497e43e58a9/8f2dcd659173cf3453856cf5fce98d82ad309637ec951593bf7cb65babad43a1	5f5e4ef5-22f0-4ff5-a693-0497e43e58a9

Figure 5.3. Control Group Path Field

inherit the control group name of the process that generates the container, each container belonging to a pod inherits the same control group path.

In the end, two additional IMA template fields were added:

- The *dependencies*, to have extra assurance in the discrimination of the entry.
- The *control group path*, to be sure to analyze a pod entry and retrieve the pod UUID.

The new template descriptor *ima-cgpath* was defined, which will be explained more in detail in chapter 6. Specifically, if an entry of the ML has the dependencies field containing the orchestrator dependencies, which in the case of k3s is identified with */rancher/k3s*, and the control group path starts with *kubepods*, then the entry is related to a pod and its pod UUID can be retrieved from the control group path field.

5.4 Keylime Integration

Once the new IMA template was put in place, the next step was to make the remote attestation framework able to understand and use it. The first consideration to make is that the Keylime Verifier, Tenant, and Registrar components should run in a Kubernetes Master Node, while the Keylime Agent should run in a Kubernetes Worker Node, as explained in section 5.5. As a reminder, the execution flow provided by Keylime is represented by the following steps:

1. First, when the Keylime Agent is kicked off, the registration protocol of the node against the Registrar component begins.
2. Once the node has proven its identity, it has to be registered against the Verifier Component, a phase that involves also the Tenant Component in the Three Party Bootstrap Key Derivation Protocol.
3. Finally, when the node is registered at the Verifier, the remote attestation can be continuously performed.

Phase 1 has not been affected by modifications, as explained in the following paragraphs, while phases 2 and 3 have been modified accordingly. Of course, in the first phase of the integration, the objective was to make the remote attestation works with the physical system taking measures with the new template, only in the end the pods' support was added.

5.4.1 Worker Node

In the attester machine, which corresponds to a Worker Node, the IMA module will be responsible to fill the ML to be sent in the Integrity Report with the new IMA cgpath template. This is the only modification required in the Keylime Agent, which simply sends the Integrity Report at each attestation cycle.

5.4.2 Master Node

In the verifier machine, which corresponds to a Master Node, the Keylime Registrar component is in charge to establish the hardware device authentication of the platform, by evaluating the genuineness of the TPM's EK. The Keylime Registrar has not been affected by modifications and performs the registration protocol as usual. When the Keylime Tenant component has to perform the registration of the host system to the Keylime Verifier, there will be an additional operation to be done: the pods' registration. Pods' registration is needed to detect if there are unknown pods running in the node, hence if only expected pods are running and in the expected way. The registration happens by providing the list of the pod UUIDs which are expected to run in the node, which can be retrieved by the orchestrator. So, each specific-tenant pod running in a worker node must be put in a list and be accompanied by its associated whitelist, to be evaluated and to be assigned with a status. The status of a pod could be:

- START, if the pod is waiting for the first integrity check;
- TRUSTED, if the pod has a trusted state;
- UNTRUSTED, if the pod has an untrusted state.

In the beginning, each pod is assigned with the START state, since the transition can be done only by the Keylime Verifier component during the evaluation process. The latter will be responsible to validate the trustworthiness of the pods, exactly with the same logic as the host machine. After having established the integrity of the ML against PCR 10, the validation of the host system happens together with the validation of the pods. The main logic related to pod follows this reasoning:

- Once established that the ML entry is related to a pod, the pod UUID is retrieved from the control group path field.
- If the pod UUID is not belonging to the list of registered pods, then the host system is evaluated untrusted and the validation stops because an unknown pod is performing executions.
- If the pod UUID is present in the list, then its whitelist is retrieved and its measures are evaluated one by one, as it happens with the host. In the end, each pod status can pass either to trusted or untrusted. If a registered pod is found untrusted, then the host system stays trusted, in order to avoid the whole platform to be restarted.

Each possible outcome of the remote attestation process will be explained in detail in the next chapter.

5.5 Architecture of the Proposed Solution

In the Kubernetes architecture, the main distinction is between master and worker nodes, as stated in 2.4. For this reason, the idea behind the thesis work was to try to integrate directly in the Kubernetes control plane the Keylime components involved in the remote attestation process. The overall architecture can be seen in figure 5.4, where the blue lines indicate the communication of the Kubernetes components, while the green ones are related to the Keylime framework. For simplicity, in the figure are pictured only one master and one worker node. The overall process can be seen as three separate phases:

- an initial one in which is performed the *set-up of the cloud environment*;
- a second phase regarding the *periodic remote attestation*;
- a third phase concerning the *life cycle management.*

During the initial phase, the needed worker nodes are instantiated by a master node to run the pods specified by the user, and the Keylime Agent is registered at the Keylime Registrar. Once the cluster is active and properly running, the periodic remote attestation can be launched by the user by sending an attestation request to the Keylime Tenant, which will provide the registration of the Agent at the Keylime Verifier. Having all the needed data, the Keylime Verifier can proceed with the periodic remote attestation. Finally, the third phase begins and from this point on, different kinds of situations can arise, as explained in section 5.5.3.

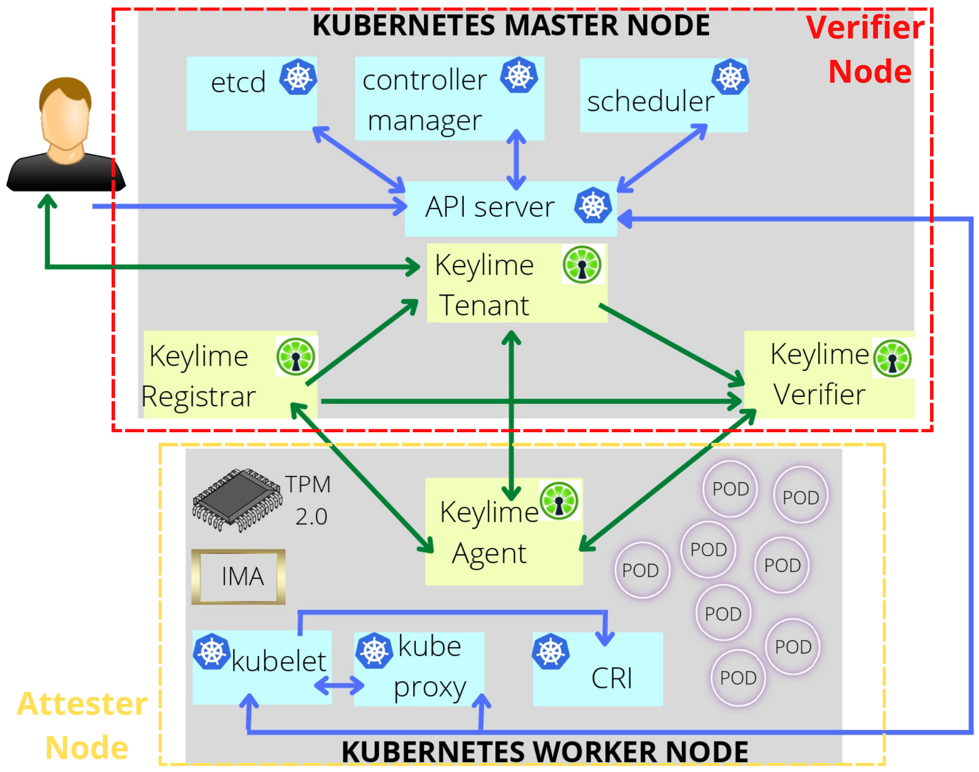


Figure 5.4. Architecture of the proposed solution

5.5.1 Setting up the Cloud Environment

The initial phase of the process can be seen in figure 5.5. When a remote user specifies a new deployment to the API server (1), the desired state is stored inside the etcd database (2), the controller manager instantiates the needed controllers, and the scheduler selects the physical nodes in which the pods should run (3). For simplicity of explanation, let us assume that only one worker node is selected by the scheduler. When the worker node is chosen, kubelet communicates with CRI which will instantiate the pods through a container runtime (4). At this point, the cluster is ready and working to maintain the actual state to match the desired one.

At the same time which the worker node was powered on, the Registration Protocol between the Keylime Agent and the Keylime Registrar starts (1), in order to enable the hardware device authentication offered by the TPM. The Registration Protocol has not undergone any changes, as stated in paragraph 5.4. On the other hand, the Registration of the Keylime Agent to the Keylime Verifier is the first step to consent the remote attestation, and it happens when the remote user wants to retrieve the integrity proof of the worker node running its pods (5). The latter operation needs to be integrated to support the pod registration, specifically by using a pod list containing the expected UUIDs and the associated whitelists, one for each pod, as explained in the following section.

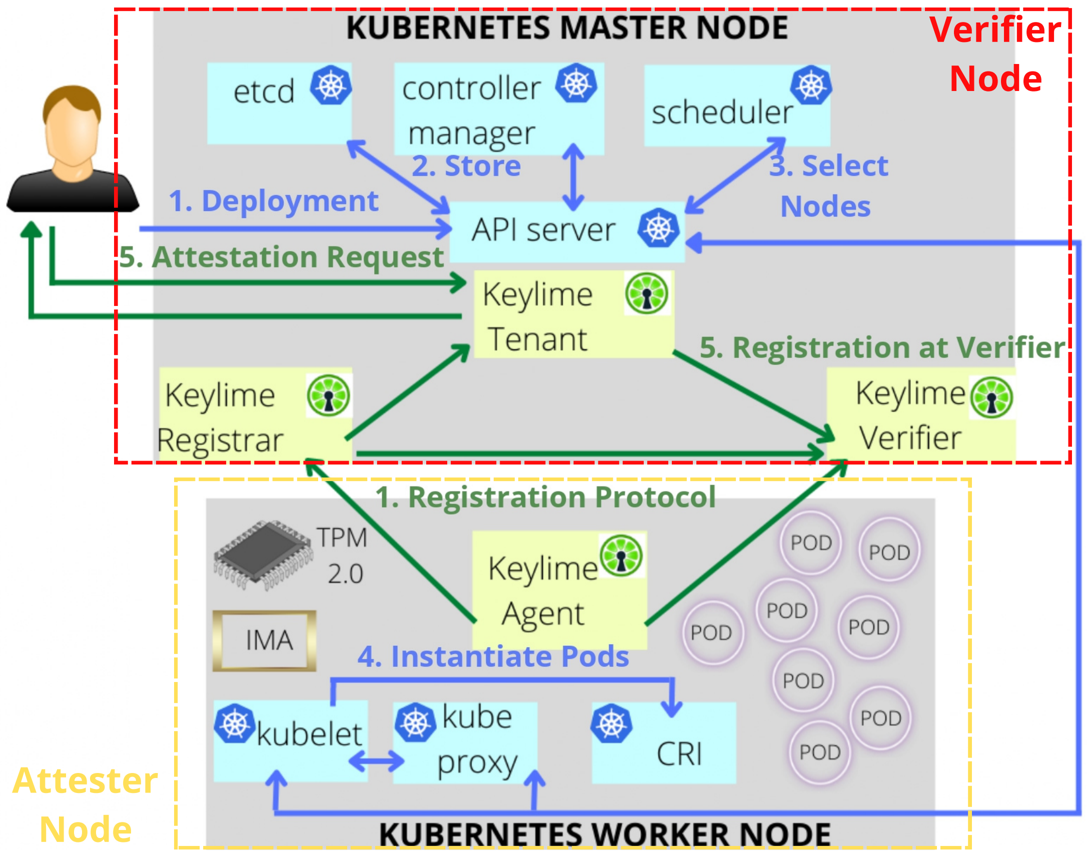


Figure 5.5. Set Up of the Cloud Environment

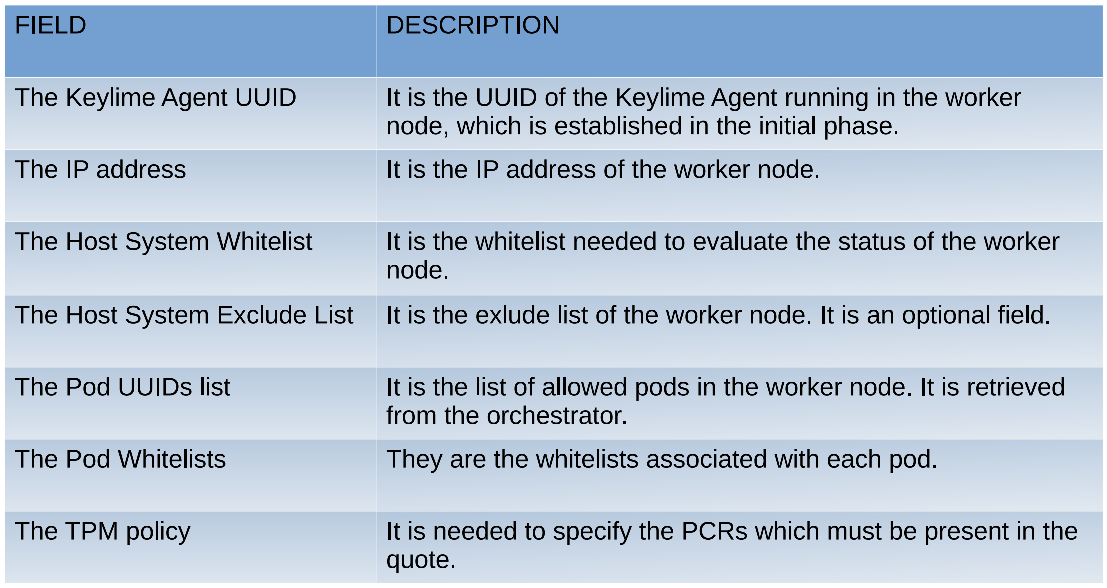
5.5.2 Periodic Remote Attestation

After the conclusion of the preliminary phase, when the user wants to know the integrity status of its pods, it communicates with the Keylime Tenant and the Registration of the Keylime Agent to the Keylime Verifier happens. The user has first to contact the orchestrator to retrieve its pod identifiers since they are dynamic objects that could change over time. As an example, the worker node can be subject to a failure and switched off: as a consequence, the pods will be rescheduled with different pod UUIDs and may run in one or more different worker nodes. After having retrieved the pod UUIDs, the user contacts the Keylime Tenant to make its Attestation Request, providing the data shown in table 5.6.

Workflow

When the Keylime Tenant has retrieved from the User the needed data to be forwarded to the Verifier, the workflow is the one shown in figure 5.7:

1. The user makes an Attestation Request to the Keylime Tenant, providing all the needed data.
2. The Three Party Bootstrap Key Derivation protocol starts, so the Keylime Tenant creates and splits the bootstrap key Kb into U and V, sending V to



FIELD	DESCRIPTION
The Keylime Agent UUID	It is the UUID of the Keylime Agent running in the worker node, which is established in the initial phase.
The IP address	It is the IP address of the worker node.
The Host System Whitelist	It is the whitelist needed to evaluate the status of the worker node.
The Host System Exclude List	It is the exlude list of the worker node. It is an optional field.
The Pod UUIDs list	It is the list of allowed pods in the worker node. It is retrieved from the orchestrator.
The Pod Whitelists	They are the whitelists associated with each pod.
The TPM policy	It is needed to specify the PCRs which must be present in the quote.

Figure 5.6. Data provided in the Attestation Request

the Keylime Verifier. The Tenant sends to the Verifier also the data received from the user, because it will be in charge to monitor the host system and the pods. At this point, the new Keylime Agent is registered at the Verifier.

3. The Keylime Agent retrieves the U part of the bootstrap key from the Tenant, together with an encrypted payload.

4. The Agent retrieves the V part of the bootstrap key from the Verifier, to recompose the bootstrap key and be able to decrypt the securely exchanged payloads received from the Tenant.

5. The Verifier starts the continuous remote attestation, providing a nonce and a policy, and receiving the integrity report.

6. The user retrieves the attestation results.

5.5.3 Life Cycle Management

Once phases 1 and 2 are completed, the periodic remote attestation is enabled and performed continuously. However, after these two phases, some kind of event can happen during the life cycle of the nodes. For instance:

- a Worker Node may be shut down, causing the death of the pods within it. Each pod has to be rescheduled;

- a new pod can be added to a Worker Node, so it has to be registered in the list and its associated whitelist should be provided;

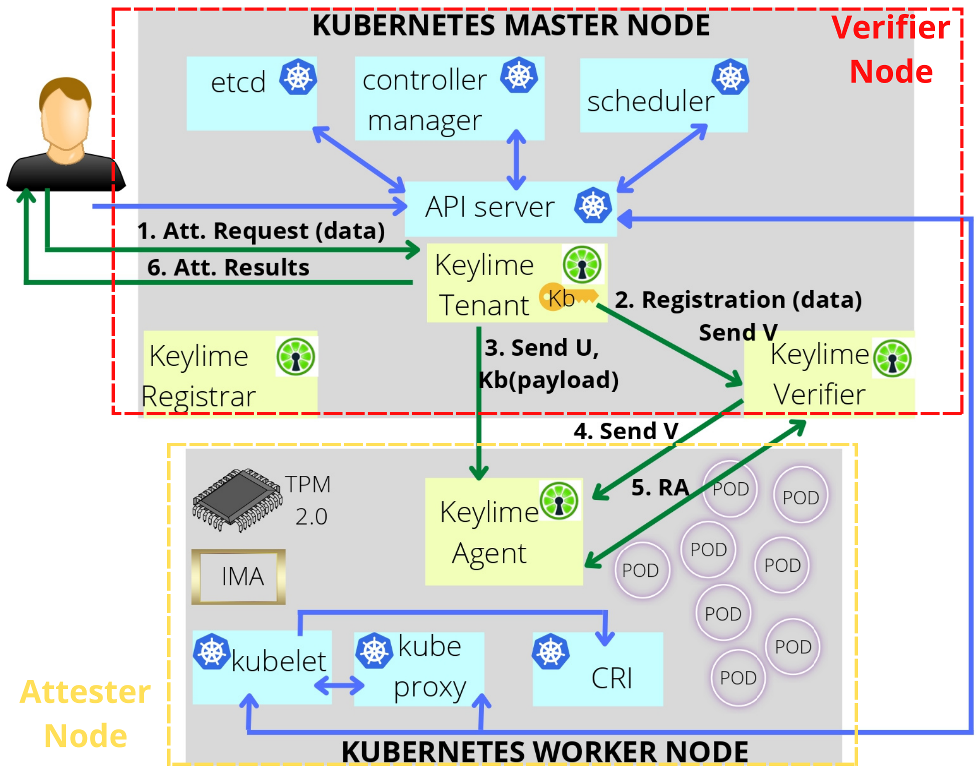


Figure 5.7. Workflow of the proposed solution

- the whitelist of a pod can be subject to an update to allow new kinds of operations;
- also the whitelist of the host can be updated.

Except for the last point, any of these events fall into the category of the migration problem, which considers that at a certain point in time a node can run dynamically any of the possible pods, according to the Kubernetes scheduler decisions. How these situations can be handled is reported in the next section 5.6.

5.6 Application of the Solution to Multiple Nodes

It is worth notice that the solution described in the previous sections was developed having in mind exactly a simple scenario: each tenant has running pods in a unique worker node, and each tenant has to retrieve its specific pod UUID list from the orchestrator. However, these first assumptions were too simplified to be used as they were in a real-world scenario. As already mentioned, when a worker node is switched off, every pod which was running within it is rescheduled with a different pod UUID. If any pod is found untrusted, then it has to be isolated and replaced with a new, fresh, and uncompromised version, which will have, again, a different pod UUID. Moreover, pods to be rescheduled can be potentially instantiated in a different worker node than the original one, based on the currently available resources. In the real case scenario, each tenant has running pods in more than

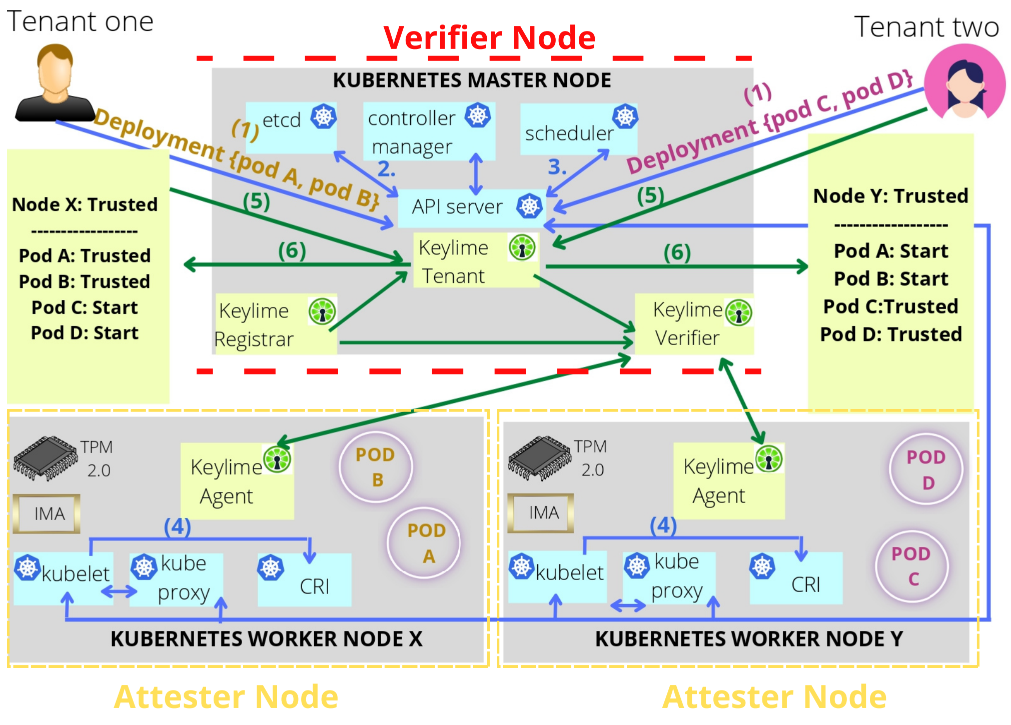


Figure 5.8. Kubernetes Cluster with two worker nodes. Each tenant is executing its pods within a single node.

one worker node, and, in each worker node, are running pods belonging to different tenants. Due to this complexity, an actual integration in the Kubernetes Control Plane is needed, which will be part of the future work, as explained in chapter 8. To overcome these limitations, at this point of the development and contextually with the k3s cluster set-up, was decided to add the assumption that within each worker node of the Kubernetes cluster are allowed all the possible pods, for each tenant, hence all the possible pods are registered in each Master Node. This choice was made for simplicity, because the Kubernetes scheduler can arbitrarily change the worker node in which a pod is executed, bringing, in fact, migration issues.

The final version of the solution was applied and tested in a cluster architecture represented in figure 5.8 and figure 5.9: for instance, if the tenant *one* has pods A and B, while tenant *two* has pods C and D, then in a master node which is running the Keylime Verifier, the pod list will contain pods A, B, C, and D and there will be all their related whitelists. Instead, in a worker node that is running the Keylime Agent, will be possible and allowed the presence of pods A, B, C, and D. Of course, it is possible that none of the pods between A, B, C, and D are executing in that specific node, but this does not affect drastically the overall performance, since each pod will remain with state START, as shown in figures. Remaining in START state means that they are not evaluated, because the first evaluation of a pod entry is performed when that specific pod UUID is found in the Measurement Log. If the pod is not executing in that node, the ML contained in the Integrity Report provided by the Keylime Agent cannot contain measures related to that pod.

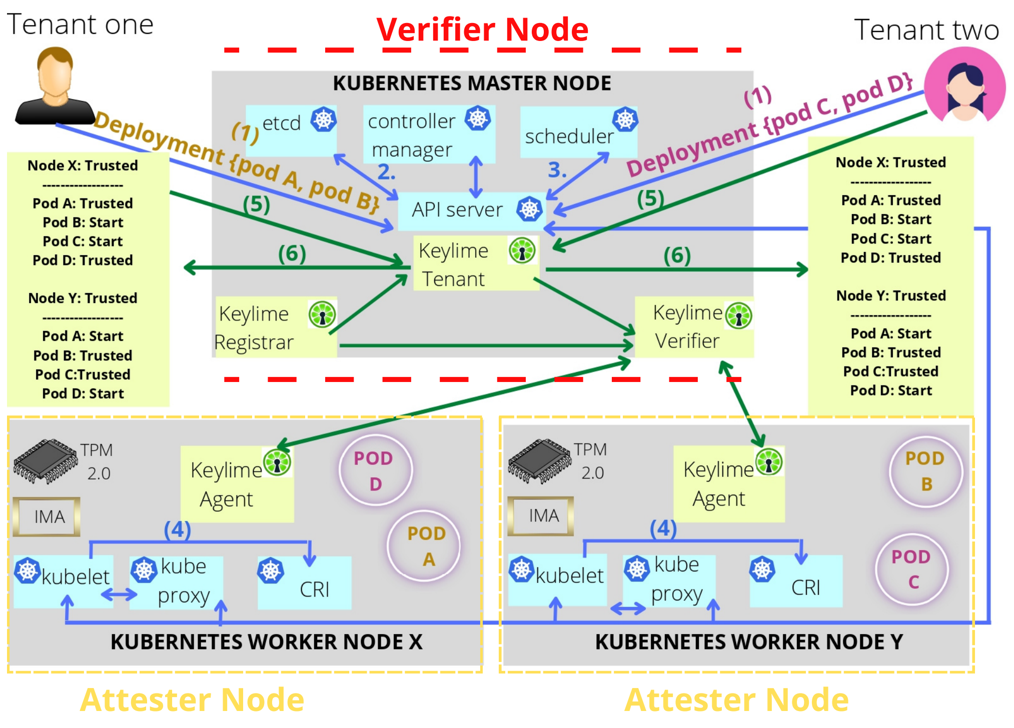


Figure 5.9. Kubernetes Cluster with two worker nodes. Each tenant is executing its pods within more nodes.

The workflow is the following:

1. The user specifies a new deployment.
2. The deployment is stored in etcd and the controllers instantiated.
3. The scheduler selects the worker nodes to be used. In figure 5.8 is selected node X for tenant one and node Y for tenant two, while in figure 5.9 both nodes are selected for both tenants.
4. The pods are instantiated through a container runtime.
5. The users ask a Remote Attestation Request.
6. The users retrieve the Remote Attestation Results.

In this way, whenever a new pod has to be added, the orchestrator has just to provide its UUID and associated whitelist to the Keylime Verifier. When a Worker Node is switched off, the migration of each pod that was running within it is allowed in any of the other Worker Nodes. When a pod wants to perform a new kind of operation which was not originally planned, its whitelist can be updated on the Verifier side, without causing any problem. To forbid the execution of a previously registered pod, it is sufficient to delete its UUID and whitelist from the Verifier. In other words, the migration problem is completely solved with this assumption, but as explained in section 5.1, it lacks in terms of optimization.

5.7 Considerations

The proposed solution is not only lacking in terms of optimization, but it is not taking into account the privacy issue as well, since a tenant can retrieve the status of the pods belonging to another one. For this reason, the purpose of the future work is also to overcome this limitation, since by integrating the Keylime Tenant's APIs directly into the API server, it is possible to retrieve the specific list of the tenants' pods and physical nodes in which they are running in a dynamic way, in order to attest only those of interest and to solve dynamically also the migration issue. In this way, the orchestrator could exploit directly the Keylime Framework by providing it with the needed data related to specific-tenant pods, to consent the usage of real lists in the process.

Finally, from the point of view of the whitelists associated with pods, they could be built starting from the container images that reside within them, and memorized at the Verifier without involving the user, as long as any different operation is required on the system.

Despite of the underlined drawbacks, the proposed solution reached its main goal of be able to shut down and rescheduled only the compromised pods, without affecting the integrity of the host system which can stay on and grant all the other services evaluated as trusted.

Chapter 6

Remote attestation of a Kubernetes cluster - Implementation

This chapter exposes in detail the implementation of the proposed solution. In the first section is presented the explanation of the new proposed IMA template, which has been used in patched Linux Kernels running on Ubuntu Server 20.04 LTS. The second section exposes the changes related to the Keylime framework, where the Keylime Agent is meant to run in a Kubernetes Worker Node and the Tenant, Verifier, and Registrar components in a Kubernetes Master Node. Specifically, it was chosen Keylime version 6.3.0, in which at each attestation cycle is sent only a portion of the ML, the one not yet attested, reducing the overall latency, as observed also by the Docker Container Attestation developers. Finally, the third section provides the ML validation results, with the possible outcomes provided by the Verifier during the Remote Attestation process.

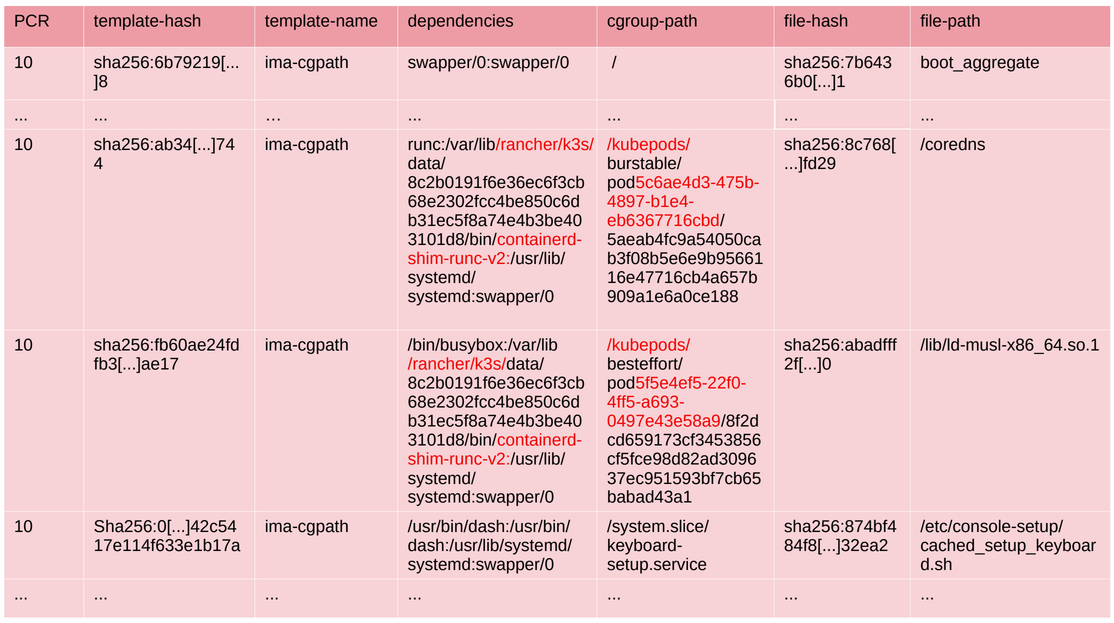
6.1 IMA Template

The new IMA template was defined in order to make possible the discrimination in the ML between an entry related to the host system and an entry related to a pod. The new template descriptor is called ima-cgpath with a format string "dep—cg-path—d-ng—n-ng". As shown in figure 6.1, the template fields used are:

- *PCR* that refers to the PCR number used to extend the current entry;
- *template-hash* that contains the digest calculated over the template fields of the current entry. This digest is extended in the PCR specified in the first field, and the SHA-256 algorithm is used;
- *template-name* that specifies the template descriptor used for the entry, which in this case is ima-cgpath;
- *dependencies* that specifies the dependencies of the process that created the entry;

- *cgroup-path* that specifies the control group path;
- *file-hash* that contains the digest of the file specified in the last field: it is the actual *measure* of the event. The SHA-256 algorithm is used.
- *file-path* that contains the file pathname.

The fields PCR, template-hash, and template-name are the mandatory ones, while the field dependencies and cgroup-path have been introduced. It is worth notice



PCR	template-hash	template-name	dependencies	cgroup-path	file-hash	file-path
10	sha256:6b79219[...]8	ima-cgpath	swapper/0:swapper/0	/	sha256:7b6436b0[...]1	boot_aggregate
...	...	...	...	...	...	...
10	sha256:ab34[...]744	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	sha256:8c768[...]fd29	/coredns
10	sha256:fb60ae24fdfb3[...]ae17	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:abadfff2f[...]0	/lib/ld-musl-x86_64.so.1
10	Sha256:0[...]42c5417e114f633e1b17a	ima-cgpath	/usr/bin/dash:/usr/bin/dash:/usr/lib/systemd/systemd:swapper/0	/system.slice/keyboard-setup.service	sha256:874bf484f8[...]32ea2	/etc/console-setup/cached_setup_keyboard.sh
...	...	...	...	...	...	...

Figure 6.1. Example of ML with ima-cgpath template

that the algorithm used for the template-hash field is SHA-256, while the IMA module uses by default SHA-1. This result has been achieved by the Docker Container Attestation developers and it has been exploited also in this solution. The latter is a flexible choice since it is possible to set the algorithm back to SHA-1 through the *ima_template_hash* kernel parameter.

By looking if the field *cgroup-path* starts with the string */kubepods* is possible to determine uniquely if the entry has been generated by the host system or a pod. The pod UUID needed to evaluate its status can be retrieved from the same field. The *dependencies* of the process that generates the entry can be used to provide extra assurance of its origin. In fact, if the orchestrator dependency identified with */rancher/k3s* is present then the entry is related to a pod, and the container runtime used to launch it can be retrieved. By default k3s uses containerd, as stated in section 2.3.6 and as shown in figure 6.1. The implementation details of the new template can be found in the Developer's manual, section B.1.

6.2 Keylime Modifications

The Keylime modifications concerned two main aspects. The first one was to make the framework understand and use the new IMA template, while the second was about adding the support needed to make and handle the pod registrations at the Verifier. Then, also the validation of the IR makes necessary changes in the Verifier. It is worth notice that the overall architecture has not been modified, rather the components Keylime Agent, Tenant, and Verifier have been integrated with new functionalities, as explained in the following sections.

6.2.1 Keylime Agent

The Keylime Agent is in charge of periodically sending an Integrity Report to the Keylime Verifier. The Integrity Report contains also the Measurement Log, so the code portion affected by modifications is the one that is handling the interaction with the IMA module. Specifically, a new class called ima-cgpath was added, containing the fields explained in section 6.1, and used to fill the ML in case the new template ima-cgpath is used.

6.2.2 Keylime Tenant

The interaction between the user and the Keylime Tenant to consent the registration at the Keylime Verifier and the related attestation procedure can be done either via Command Line Interface or REST APIs. The modifications needed were related to the macro-operations:

- *status*, to retrieve the operational state of the Keylime Agent (i.e. the Kubernetes Worker Node) and the pods' states running within it. The pods' states are explained in 5.4.2, while the possible operational states of the Keylime Agent could be seen in table 6.2.
- *add*, to register a new Keylime Agent at the Verifier with the additional data of pods list and pod whitelists. The whole data to be provided to perform this operation is shown in table 5.6.
- *update*, to update an already registered Keylime Agent at the Verifier, of course, with different data than the one passed during the *add* operation.
- *delete*, to delete an Agent from the Keylime Verifier.

Specifically, some new REST APIs have been added and some have been modified accordingly. The detail of the implementation can be retrieved in the User's manual, section A.7.

STATE	DESCRIPTION
Registered	if the Agent has only been registered at the Registrar with Registration Protocol
Start	if the system is not yet attested
Saved	if the first attestation is started but not yet completed
Get Quote	if the previous evaluations are completed and the system is trusted
Get Quote Retry	if the last TPM quote has not been received by Verifier
Provide V	if the Agent is retrieving the V part of the Bootstrap Key from Verifier
Provide V Retry	if the Agent is trying again to retrieve the V part of the Bootstrap Key from Verifier
Failed	if some kind of failure occurred during the process
Terminated	if the attestation procedure has been stopped
Invalid Quote	if the system has been found untrusted in the last evaluation
Tenant Failed	if the Tenant component has failed

Figure 6.2. Possible Operational States of the Keylime Agent

6.2.3 Keylime Verifier

The Keylime Verifier has to be modified in order to be able to understand and validate the new data received from the Keylime Tenant. Specifically, at the moment of the *add* and *update* operations, it fills a Database in which the Keylime Agent pods list and whitelists have to be stored together with the usual data. Specific-Agent data stored within the Database has to be removed when the *delete* operation is performed. At the moment of the *status* operation, the Verifier has to retrieve the operational state of the specified Keylime Agent and related pods.

The Keylime Verifier uses a new IMA validator for the class ima-cgpath, which was needed to check the integrity of the whole ML against IMA PCR. After the integrity validation of the whole ML, the Keylime Verifier proceeds with the validation of each ML entry, using the host whitelists and the pods' whitelists accordingly. All the possible outcomes given by the Verifier are explained in the following sections.

6.3 Remote Attestation Outcomes

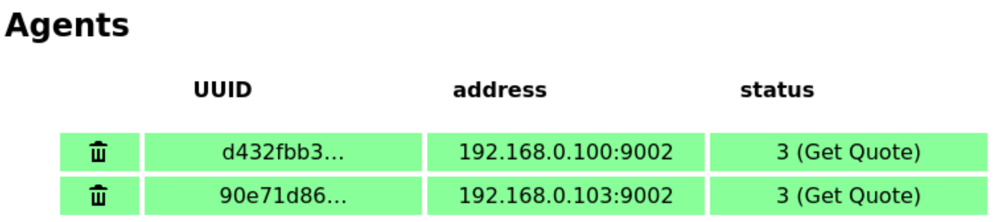
This section provides the possible results of the remote attestation process applied to a Kubernetes Worker Node.

6.3.1 Attester system evaluated as trusted

Whether the attester has registered pods or not, it is evaluated as trusted by the Keylime Verifier if:

- the quote is fresh and the signature is valid;
- all the PCR provided in the mask are present in the quote;
- the ML is not tampered with and the measures match the ones in the whitelist;

while in the specific case of registered pods, the attester system stays trusted if the previous constraints remain satisfied and in the ML are not detected unregistered pods. In figure 6.3, can be seen a screenshot taken from the Keylime web application picturing two attesters evaluated trusted (i.e. with the operational state set to *Get Quote*).



Agents

	UUID	address	status
🗑	d432fbb3...	192.168.0.100:9002	3 (Get Quote)
🗑	90e71d86...	192.168.0.103:9002	3 (Get Quote)

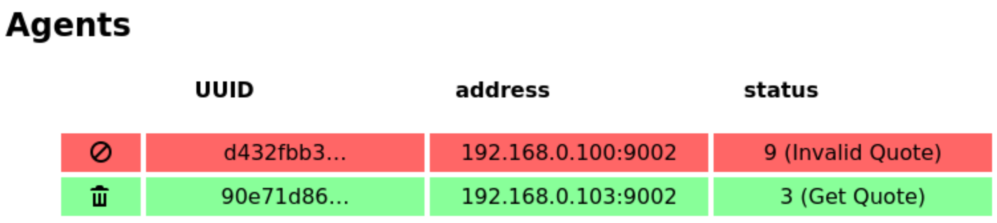
Figure 6.3. Two Attesters Evaluated Trusted

6.3.2 Attester system evaluated as untrusted

Whether the system to be evaluated has registered pods or not, it is evaluated as untrusted by the Keylime Verifier if:

- the quote is not fresh or the signature is not valid;
- one or more PCR provided in the mask is not present in the quote;
- the validation of ML integrity fails or one or more measures does not match the ones present in the whitelist.

If in the attester system there are registered pods, it is evaluated as untrusted if in the ML are present unregistered pods. In figure 6.4 can be seen a screenshot taken



Agents

	UUID	address	status
⊘	d432fbb3...	192.168.0.100:9002	9 (Invalid Quote)
🗑	90e71d86...	192.168.0.103:9002	3 (Get Quote)

Figure 6.4. One Attester Evaluated Untrusted

from the Keylime web application picturing two attesters, one evaluated trusted and the other untrusted (i.e. with the operational state set to *Invalid Quote*). As an example, if the attester is evaluated untrusted because one measure of the ML does not match its whitelist, from the log file of the verifier a message like this can be retrieved:

```
Warning: Hashes for file /usr/bin/getent don’t match
   3e774af1026aeac946858c0b6d99ec7e423fdd2d7dca2267b4b203fd4a2cb8da
```

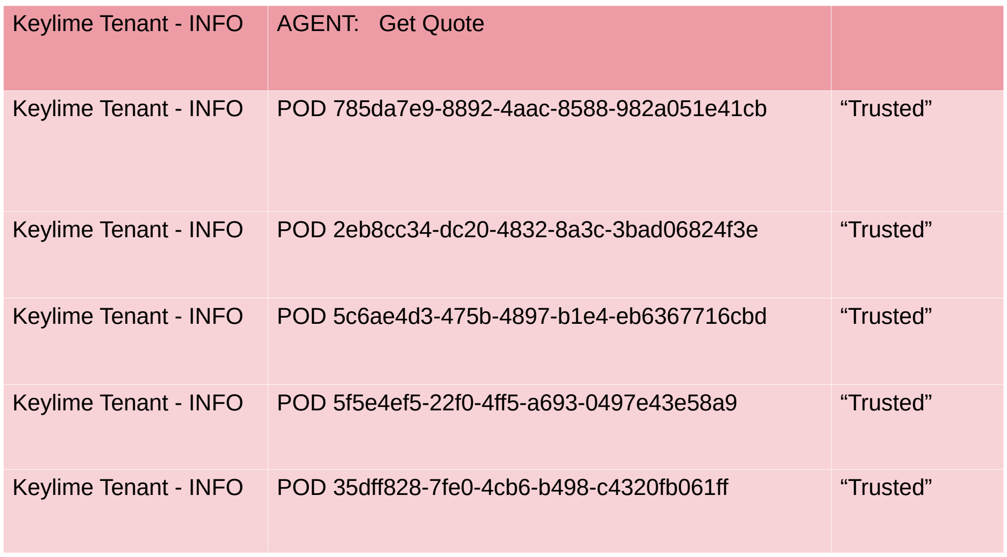
while if the reason is an unregistered pod:

```
Error: Unknown pod found in Measurement List, pod ID:
   35dff828-7fe0-4cb6-b498-c4320fb061ff
```

6.3.3 Attester system evaluated as trusted with registered pods

While the Keylime Verifier is establishing the ML integrity, it can associate the entry either to the attester or to a pod, by looking at the *dependencies* and *cgroup-path* fields. If the *dependencies* field contains a container runtime and the *cgroup-path* field contains a pod identifier, then the entry is related to a pod and it can be evaluated against a pod-specific whitelist. The process of evaluation is exactly the same as the attester system and its own whitelist. When the validation of the attester system gives a positive result, it is evaluated as trusted, and there are three possible cases related to the registered pods, each one explained in the following sections.

Each registered pod is trusted



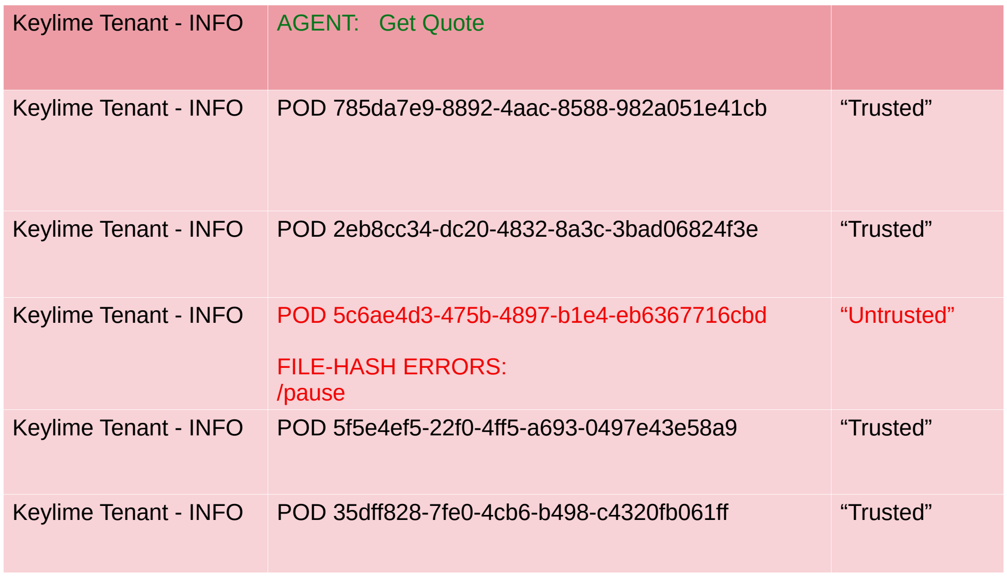
Keylime Tenant - INFO	AGENT: Get Quote	
Keylime Tenant - INFO	POD 785da7e9-8892-4aac-8588-982a051e41cb	“Trusted”
Keylime Tenant - INFO	POD 2eb8cc34-dc20-4832-8a3c-3bad06824f3e	“Trusted”
Keylime Tenant - INFO	POD 5c6ae4d3-475b-4897-b1e4-eb6367716cbd	“Trusted”
Keylime Tenant - INFO	POD 5f5e4ef5-22f0-4ff5-a693-0497e43e58a9	“Trusted”
Keylime Tenant - INFO	POD 35dff828-7fe0-4cb6-b498-c4320fb061ff	“Trusted”

Figure 6.5. The attester system and each registered pod are trusted

When the evaluation process finishes for each registered pod, the best outcome is represented by having all of them as trusted. In figure 6.5 can be seen the output related to this case displayed by the tenant Command Line Interface when the command *status* is asked. As the attester system, a pod is evaluated as trusted if each related measure present in the ML belongs to its whitelist and its value is among those expected.

The pod measure does not match the pod whitelist

Another outcome is that a pod is found untrusted. One of the possible reasons is that the measured file contained in the ML is present in its whitelist, but its hash value does not match any of those expected. In this case, the output reported by the Verifier will indicate the files which do not match the measure under the string "FILE-HASH ERRORS", as shown in figure 6.6.



Keylime Tenant - INFO	AGENT: Get Quote	
Keylime Tenant - INFO	POD 785da7e9-8892-4aac-8588-982a051e41cb	“Trusted”
Keylime Tenant - INFO	POD 2eb8cc34-dc20-4832-8a3c-3bad06824f3e	“Trusted”
Keylime Tenant - INFO	POD 5c6ae4d3-475b-4897-b1e4-eb6367716cbd FILE-HASH ERRORS: /pause	“Untrusted”
Keylime Tenant - INFO	POD 5f5e4ef5-22f0-4ff5-a693-0497e43e58a9	“Trusted”
Keylime Tenant - INFO	POD 35dff828-7fe0-4cb6-b498-c4320fb061ff	“Trusted”

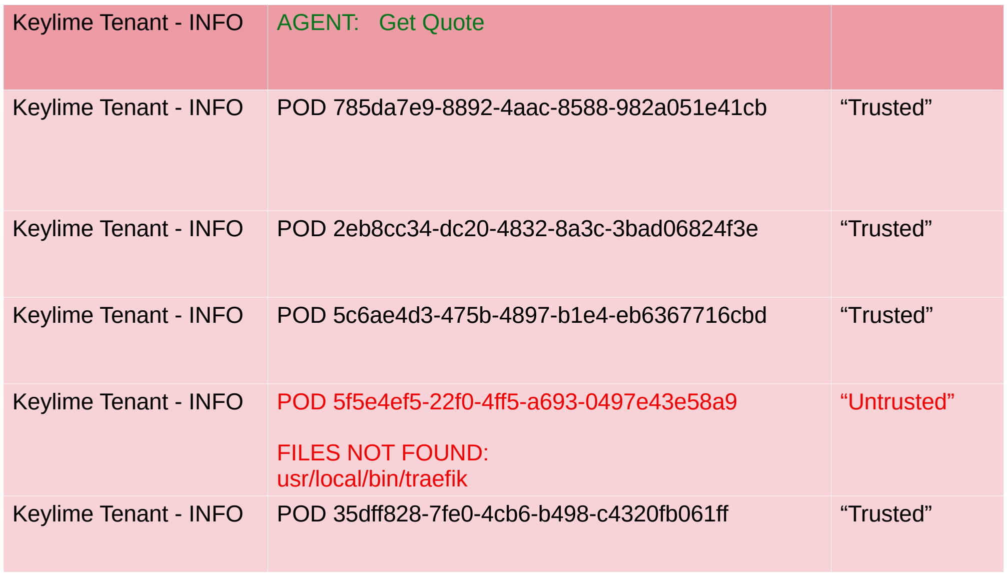
Figure 6.6. One pod is evaluated untrusted. The file /pause is present in its whitelist but the measure is not among those expected.

The measured pod file is not in the pod whitelist

The last outcome case is given when a pod is evaluated untrusted because the file is not present in its whitelist. In this case, the output reported by the Verifier will indicate the files which are not present in the whitelist under the string "FILE NOT FOUND", as shown in figure 6.7.

6.4 Considerations

The outcomes represented in figures 6.6 and 6.7 shown as the attester system stays with operational state *Get Quote*, that was the work objective. In fact, to be able to shut down and reschedule only the compromised pod, the prover system has to be considered trusted.



Keylime Tenant - INFO	AGENT: Get Quote	
Keylime Tenant - INFO	POD 785da7e9-8892-4aac-8588-982a051e41cb	“Trusted”
Keylime Tenant - INFO	POD 2eb8cc34-dc20-4832-8a3c-3bad06824f3e	“Trusted”
Keylime Tenant - INFO	POD 5c6ae4d3-475b-4897-b1e4-eb6367716cbd	“Trusted”
Keylime Tenant - INFO	POD 5f5e4ef5-22f0-4ff5-a693-0497e43e58a9 FILES NOT FOUND: usr/local/bin/traefik	“Untrusted”
Keylime Tenant - INFO	POD 35dff828-7fe0-4cb6-b498-c4320fb061ff	“Trusted”

Figure 6.7. One pod is evaluated untrusted. The file /usr/local/bin/traefik is not present in its whitelist.

These results can be obtained in two ways. The first one is by changing ”by hand” the hashes in the pod whitelists or deleting the contained files. The second way, more efficient, is to simulate a real attack against a pod, by entering it from the Worker Node and changing its binaries. The attester system stays trusted because the event ”unauthorized execution from a registered pod” is extended as well in IMA PCR, so the Keylime Verifier does not complain about the integrity status of the ML against the value of the IMA PCR.

Chapter 7

Test and Validation

This chapter provides the test and validation phase of the proposed solution. Specifically, functional tests were made to evaluate if the Keylime framework was able to correctly perform the remote attestation process, while performance tests were made to establish the timing and resources required during the process.

7.1 Testbed

The testbed put in place to test the solution was composed of three machines:

- One *Kubernetes Master Node*, running the Keylime Tenant, Registrar, and Verifier components. The OS used is Ubuntu server 20.04 LTS.
- Two *Kubernetes Worker Nodes*, running the Keylime Agent component. The OS used is Ubuntu server 20.04 LTS with a patched Linux kernel based on version 5.13 and using the new IMA template explained in section 6.1.

All the used machines are Intel NUC equipped with an Intel Core i5-5300U Processor, 16 GB of RAM, and a TPM 2.0 chip.

The appendix A shows the steps to configure the testbed and to turn it into a k3s cluster.

7.2 Functional Tests

Functional tests were made to evaluate if the proposed solution works as expected. As a reminder, in each Worker Node are registered all the allowed pods, hence a pod could have:

- operational state set to 0, which corresponds to START;
- operational state set to 1, which corresponds to TRUSTED;

- operational state set to 2, which corresponds to UNTRUSTED.

All pods running on a Worker Node other than the one being evaluated will keep their operational state set to 0.

The cases treated in functional tests include:

- tests with trusted node and pods;
- tests with trusted node and untrusted pod;
- tests with untrusted node.

7.2.1 Tests with trusted node and pods

If the host system and each pod running within it match the corresponding whitelist values, it is expected that the Keylime Verifier will evaluate as trusted both the host system and each pod.

On one of the Kubernetes Worker Nodes, the Keylime Agent has been launched with UUID "d432fbb3-d2f1-4a97-9ef7-75bd81c00000", which is the default one, and the node has been labeled with the command:

```
kubectl label nodes torsec-k3s-nuc-01 disktype=ssd
```

The label was needed just to force the Kubernetes Scheduler to launch the pods exactly in this specific Worker Node and not in the other one. The attester system has been provided with 5 pod replicas with base image Nginx, through the file nginx-deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
```

```
        - containerPort: 80
      nodeSelector:
          disktype: ssd
```

As explained in A.3, once the registration phase is completed, the Keylime Verifier starts the RA. By interacting with the Tenant Web app with a GET request at `https://<Verifier_ip_ address>:<Tenant_Webapp_port>/agents/ d432fbb3-d2f1-4a97-9ef7-75bd81c00000`, the following response is provided:

```
"code": "200"
"status": "Success"
"results": {
    "operational_state": "3"
    "v": "Sh9TjR38VTeW8H9ZQk0HSRKCsoZjPqe/6a/RVS4dkFw="
    "ip": "192.168.0.100"
    "port": "9002"
    ...
    "pods": [
            {
            "pod": "58164ca4-f0b8-49fc-9067-3ed46a98d9a1",
            "operational_state": "0"
            "allowlist": {...}
            "exclude": []
            "fnf": []
            "filehash_err": []
            },
            {
            "pod": "226aed86-763b-4a3e-925b-82e50146171e"
            "operational_state": "1"
            "allowlist": {...}
            "exclude": []
            "fnf": []
            "filehash_err": []
            },
            {
            "pod": "bead1494-a2ff-4b2b-bead-97f911a0039f"
            "operational_state": "1"
            "allowlist": {...}
            "exclude": []
            "fnf": []
            "filehash_err": []
            },
            {
            "pod": "b50d69cd-1ce9-4f4b-a577-3d87328c9810"
            "operational_state": "1"
            "allowlist": {...}
            "exclude": []
            "fnf": []
            "filehash_err": []
            },
            {
            "pod": "27d3b7c7-c23c-4e6d-a46c-0ac8c9be7ec1"
            "operational_state": "1"
            "allowlist": {...}
            "exclude": []
            "fnf": []
            "filehash_err": []
            },
            {
            "pod": "e4e20e81-9fe0-4ab5-832b-0a7230bca31e "
            "operational_state": "1"
            "allowlist": {...}
            "exclude": []
            "fnf": []
            "filehash_err": []
            },
            {
            "pod": "dd5e909a-f74a-407d-99a5-1f97020099b8"
            "operational_state": "1"
            "allowlist": {...}
            "exclude": []
            "fnf": []
            "filehash_err": []
            },
            ]
    "id": "d432fbb3-d2f1-4a97-9ef7-75bd81c00000"
    }
```

where:

- the operational_state of the host system is set to 3, which corresponds to GET QUOTE;
- the operational_state of each pod running within agent d432fbb3-d2f1-4a97-9ef7-75bd81c00000 is set to 1, which corresponds to TRUSTED;

as expected. The only pod with operational_state equal to 0, which corresponds to START, is the one with UUID 58164ca4-f0b8-49fc-9067-3ed46a98d9a1, since it is running in the other Worker Node.

As shown, the Keylime Verifier was able to evaluate correctly the integrity of the Worker Node.

7.2.2 Tests with trusted node and untrusted pod

As a first test, let us execute in an Nginx pod some software that is not part of its whitelist. In the examined Worker Node, open a bash on one of the Nginx pods running within it through the command:

```
k3s kubectl exec  <pod_name> -n <namespace> -it -- bash
```

and then launch the command ls. Since the Nginx associated whitelist allows neither the bash nor the ls command, what is expected is that the Keylime Verifier will evaluate untrusted the pod, while the host system will remain in GET QUOTE.

By interacting with the Tenant Web app with a GET request at `https://<Verifier_ip_address>:<Tenant_Webapp_port>/agents/d432fbb3-d2f1-4a97-9ef7-75bd81c00000`, the following response is provided:

```
"code": "200"
"status": "Success"
"results": {
    "operational_state": "3"
    "v": "Sh9TjR38VTeW8H9ZQk0HSRKCsoZjPqe/6a/RVS4dkFw="
    "ip": "192.168.0.100"
    "port": "9002"
    ...
    "pods": [
            {
            "pod": "58164ca4-f0b8-49fc-9067-3ed46a98d9a1",
            "operational_state": "0"
            "allowlist": {...}
            "exclude": []
            "fnf": []
            "filehash_err": []
            },
            {
            "pod": "226aed86-763b-4a3e-925b-82e50146171e"
            "operational_state": "1"
            "allowlist": {...}
            "exclude": []
            "fnf": []
            "filehash_err": []
            },
            {
            "pod": "bead1494-a2ff-4b2b-bead-97f911a0039f"
            "operational_state": "1"
            "allowlist": {...}
            "exclude": []
            "fnf": []
            "filehash_err": []
            },
            {
            "pod": "b50d69cd-1ce9-4f4b-a577-3d87328c9810"
            "operational_state": "1"
            "allowlist": {...}
            "exclude": []
            "fnf": []
            "filehash_err": []
            },
            {
            "pod": "27d3b7c7-c23c-4e6d-a46c-0ac8c9be7ec1"
            "operational_state": "2"
            "allowlist": {...}
            "exclude": []
            "fnf": [
              "/bin/bash/",
              "/lib/x86_64-linux-gnu/libtinfo.so.5.9",
              "/bin/ls",
              "/lib/x86_64-linux-gnu/libselinux.so.1",
            ],
            "filehash_err": []
            },
            {
            "pod": "e4e20e81-9fe0-4ab5-832b-0a7230bca31e "
            "operational_state": "1"
            "allowlist": {...}
            "exclude": []
            "fnf": []
            "filehash_err": []
            },
            {
            "pod": "dd5e909a-f74a-407d-99a5-1f97020099b8"
            "operational_state": "1"
            "allowlist": {...}
            "exclude": []
            "fnf": []
            "filehash_err": []
            },
            ]
    "id": "d432fbb3-d2f1-4a97-9ef7-75bd81c00000"
    }
```

where the compromised pod has the operational state set to 2 (UNTRUSTED), its ”files not found” list has the unrecognized measured files, and the host operational state is set to 3 (GET QUOTE), as expected.

Now, let us add a new entry within the Nginx pod whitelist: a file called wrong_hash, which corresponds to the measure of some script. Since the file /bin/bash/wrong_hash is measured, by executing it with different content, what is expected is that the file will be put within the list ”file hash error” and the pod considered as untrusted.

By interacting with the Tenant Web app with a GET request at `https://<Verifier_ip_address>:<Tenant_Webapp_port>/agents/d432fbb3-d2f1-4a97-9ef7-75bd81c00000`, the following response is provided:

```
"code": "200"
```

```
"status": "Success"
"results": {
    "operational_state": "3"
    "v": "Sh9TjR38VTeW8H9ZQk0HSRKCsoZjPqe/6a/RVS4dkFw="
    "ip": "192.168.0.100"
    "port": "9002"
    ...
    "pods": [
            {
            "pod": "58164ca4-f0b8-49fc-9067-3ed46a98d9a1",
            "operational_state": "0"
            "allowlist": {...}
            "exclude": []
            "fnf": []
            "filehash_err": []
            },
            {
            "pod": "226aed86-763b-4a3e-925b-82e50146171e"
            "operational_state": "1"
            "allowlist": {...}
            "exclude": []
            "fnf": []
            "filehash_err": []
            },
            {
            "pod": "bead1494-a2ff-4b2b-bead-97f911a0039f"
            "operational_state": "1"
            "allowlist": {...}
            "exclude": []
            "fnf": []
            "filehash_err": []
            },
            {
            "pod": "b50d69cd-1ce9-4f4b-a577-3d87328c9810"
            "operational_state": "1"
            "allowlist": {...}
            "exclude": []
            "fnf": []
            "filehash_err": []
            },
            {
            "pod": "27d3b7c7-c23c-4e6d-a46c-0ac8c9be7ec1"
            "operational_state": "2"
            "allowlist": {...}
            "exclude": []
            "fnf": [
              "/bin/bash/",
              "/lib/x86_64-linux-gnu/libtinfo.so.5.9",
              "/bin/ls",
              "/lib/x86_64-linux-gnu/libselinux.so.1",
            ],
            "filehash_err": [
            "/bin/bash/wrong_hash",
            ]
            },
            {
            "pod": "e4e20e81-9fe0-4ab5-832b-0a7230bca31e "
            "operational_state": "1"
            "allowlist": {...}
            "exclude": []
            "fnf": []
            "filehash_err": []
            },
            {
            "pod": "dd5e909a-f74a-407d-99a5-1f97020099b8"
            "operational_state": "1"
            "allowlist": {...}
            "exclude": []
            "fnf": []
            "filehash_err": []
            },
            ]
    "id": "d432fbb3-d2f1-4a97-9ef7-75bd81c00000"
    }
```

where the compromised pod has operational state set to 2 (UNTRUSTED), its ”file hash error” list has the unrecognized measured file, and the host operational state is set to 3 (GET QUOTE), as expected.

7.2.3 Tests with untrusted node

Finally, either by executing an unauthorized script in the host system or by executing an unregistered pod, what is expected is that the system will get an operational state set to INVALID QUOTE. First, let us create a new deployment.yaml file with content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: alpine-deployment
spec:
  selector:
    matchLabels:
      app: alpine
  replicas: 1
  template:
    metadata:
      labels:
        app: alpine
    spec:
      containers:
      - name: alpine
        image: alpine
        ports:
        - containerPort: 10000
      nodeSelector:
        disktype: ssd
```

and then apply it with: `kubectl apply -f deployment.yaml`.
By interacting with the Tenant Web app with a GET request at `https://<Verifier_ip_address>:<Tenant_Webapp_port>/agents/d432fbb3-d2f1-4a97-9ef7-75bd81c00000`, the following response is provided:

```
"code": "200"
"status": "Success"
"results": {
    "operational_state": "9"
    "v": "Sh9TjR38VTeW8H9ZQk0HSRKCsoZjPqe/6a/RVS4dkFw="
    "ip": "192.168.0.100"
    "port": "9002"
    ...
    "pods": [...]
    "id": "d432fbb3-d2f1-4a97-9ef7-75bd81c00000"
    }
```

where the operational state of the host is set to 9 (INVALID QUOTE), as expected. The same result has been observed by executing inside the /usr/bin/ directory of the host system an arbitrary script that was not present in its whitelist, showing that the keylime Verifier evaluates correctly the integrity status of the host system and stops polling the Keylime Agent.

7.3 Performance Tests

The metrics used for the performance evaluation were:

- the time required by the Keylime Verifier for an attestation cycle;
- the CPU consumption of the Worker Node with and without the RA;
- the RAM consumption of the Worker Node with and without the RA.

The values associated with the previous metrics were measured taking into account an increasing number of pods, starting from 1 up to 110. By default, Kubernetes configures nodes on standard clusters to run no more than 110 pods: for this reason, performance beyond 110 pods was not evaluated.

In figure 7.1 is depicted the RAM consumption of the Worker Node calculated

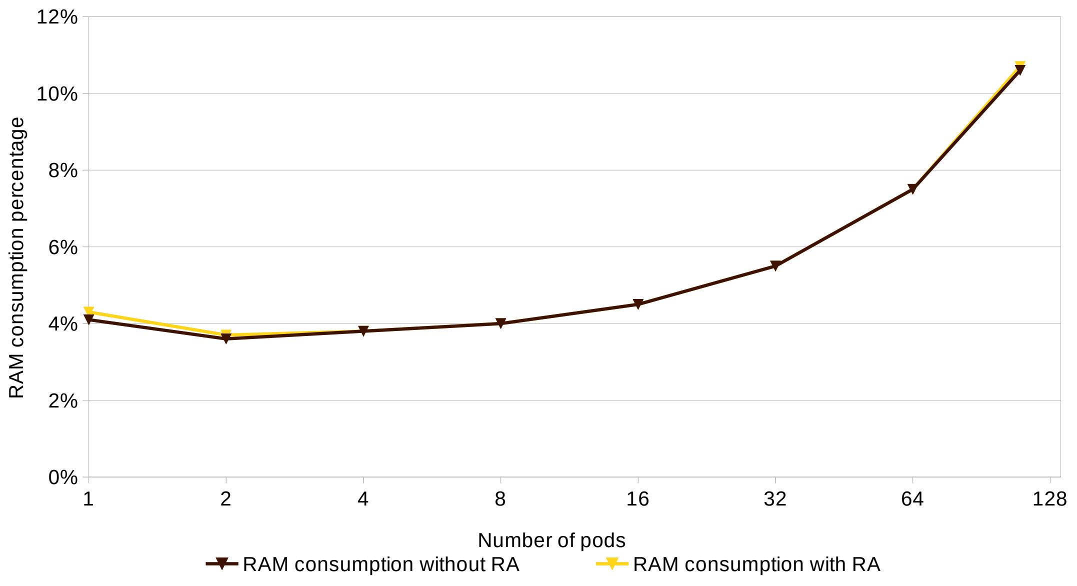


Figure 7.1. RAM consumption with and without RA

over a 10 minutes time window. As shown, the Remote Attestation procedure does not affect the overall performance, since the two graphs overlap regardless of the number of pods. The figure 7.2 shows the CPU consumption of the Worker Node calculated over a 10 minutes time window. As shown, in this case, there is an impact of almost 1,23%, which remains constant as the number of pods changes. This increase in the CPU usage is due to the Keylime Agent which is continuously polled by the Keylime Verifier to provide an Integrity Report.

Finally, in figure 7.3 are depicted the time values of the quote operation and the whole attestation procedure calculated on the average of 200 attestations. As can be seen, the quote operation requires almost all of the overall time, while time required by the whole attestation cycle includes:

- the time required by the TPM to perform the quote operation;
- the time needed to read and insert the ML file in the Integrity Report by the Keylime Agent;
- the time required to transfer the IR over the network;
- the time to verify the integrity and validate the content of the IR by the Keylime Verifier.

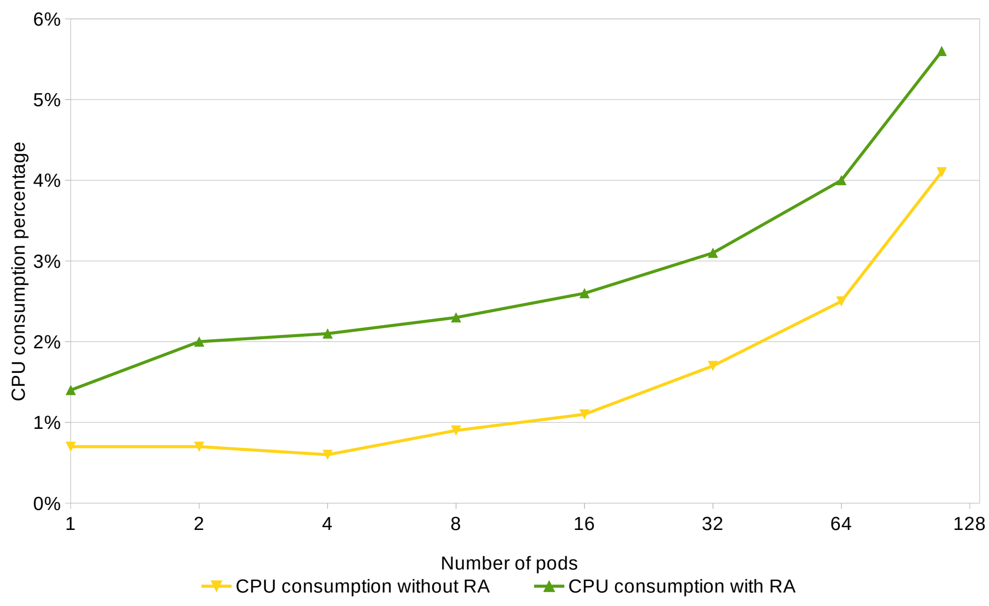


Figure 7.2. CPU consumption with and without RA

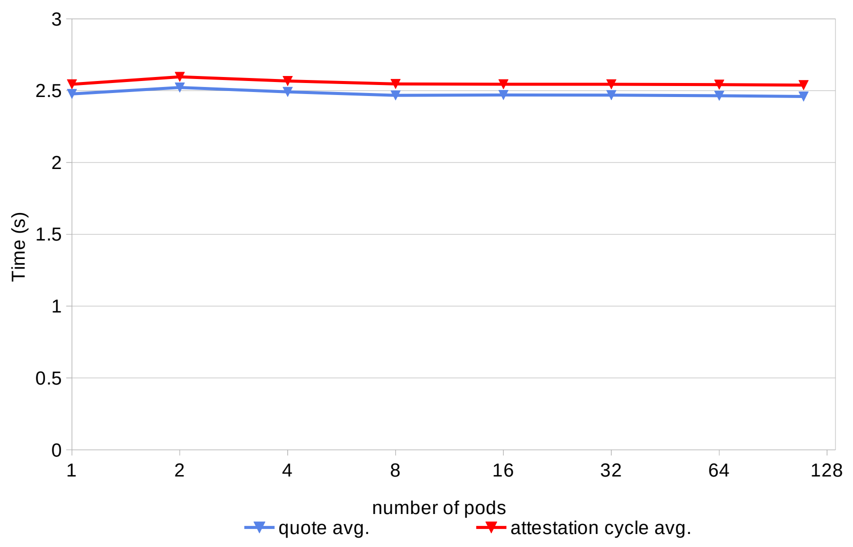


Figure 7.3. RA latency assessed as pods grow

As can be seen in the figure, both times are stable with almost 2.4 seconds for the quote operation and 2.5 seconds for all the cycle. From these results it can be deduced that the proposed solution to perform the RA of a Worker Node and the pods running within it is efficiently scalable, and, having been considered an increasing number of pods, it can be said that the overall impact is negligible.

Chapter 8

Conclusions

The purpose of this thesis work was to implement an efficient solution for allowing the remote attestation of applications deployed in pods, the smallest scheduling units in the Kubernetes ecosystem. The introduction of the new IMA template with the additional fields to uniquely identify the entries associated with pods made it possible to achieve the desired goal. Since it relies upon the TPM 2.0 specification, Keylime was the selected attestation framework for developing the solution. The latter has been modified accordingly to work with the new template and to introduce the possibility of a single-pod remote attestation. Furthermore, Keylime was also provided with additional APIs to support the registration of the pods associated with each Keylime Agent.

Previously proposed solutions, such as DIVE, Container-IMA, and Docker Container Attestation, targets the container concept and possess a series of drawbacks to be addressed. This thesis proposes a new way to allow the remote attestation of pods, considering each one of them as a separate entity and the variety of container runtimes that can be used to launch them. As demonstrated with the performance tests performed in the laboratory, the proposed solution is scalable, can be adapted with any container runtime allowed by Kubernetes, and has a negligible impact as the number of pods increases. The time required for a whole attestation cycle is heavily influenced by the time required for the quote operation, hence either by working on the TPM chip performance or using a different anchor technology, a significant improvement can be achieved. As shown in section 7.3, the CPU latency grows evenly as the number of pods increases, remaining around 1,23%, which makes the solution applicable in a real case scenario.

The developed solution represents a valuable starting point for future work, mostly because it can be easily integrated into the Kubernetes control plane. However, some challenges and criticalities remain open, which can be addressed and solved in the future by increasing the objectives to be achieved. The privacy issue in a multi-tenant cloud environment, in which a node is executing applications of different users, can be solved by making an actual integration in the Kubernetes control plane, to allow a real-time bind between the pod and the specific user who is exploiting it. The latter represents also a way both to automate the generation of the pod list for a specific user and a complete solution for the migration problem, addressing some optimization issues present in the single-tenant developed solution. Another observation for future work is that the solution is strongly linked to a specific kernel version, hence there is a need for kernel changes to the nodes of the infrastructure, which cannot always be a suitable choice. Exploring new mechanisms to identify the specific pod and the specific container runtime used to launch it, or including the IMA patch directly in the source code, is definitely a starting point to keep in mind. Despite the underlined points of reflection, the proposed solution reached its main goal regardless of the complexity of the cloud infrastructure, allowing the identification of the possible compromised portions of an application running in a cloud node and granting the services deployed in the other portions.

Bibliography

[1] Benefits of cloud computing, IBM Cloud Education, https://www.ibm.com/cloud/learn/benefits-of-cloud-computing

[2] M. Russinovich, M. Costa, C. Fournet, D. Chisnall, A. Delignat-Lavaud, S. Clebsch, K. Vaswani, and V. Bhatia, "Toward confidential cloud computing", Commun. ACM, vol. 64, may 2021, pp. 54–61, DOI 10.1145/3453930

[3] 2022 Verizon Data Breach Investigations Report, https://www.verizon.com/business/resources/reports/2022-dbir-public-sector-snapshot.pdf

[4] NIST Special Publication 800-190, Application Container Security Guide, https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-190.pdf

[5] Trusted computing by Trusted Computing Group, https://trustedcomputinggroup.org/trusted-computing/

[6] Datadog's 2021 Container Report, https://www.datadoghq.com/container-report/#1

[7] G. Blinowski, A. Ojdowska, and A. Przybylek, "Monolithic vs. microservice architecture: A performance and scalability evaluation", IEEE Access, vol. 10, 2022, pp. 20357–20374, DOI 10.1109/ACCESS.2022.3152803

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

[9] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures", 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), 2018, pp. 000149–000154, DOI 10.1109/CINTI.2018.8928192

[10] T. Yarygina and A. H. Bagge, "Overcoming security challenges in microservice architectures", 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), 2018, pp. 11–20, DOI 10.1109/SOSE.2018.00011

[11] Open Container Initiative, https://opencontainers.org/

[12] Docker - Its components and the OCI, https://accenture.github.io/blog/2021/03/18/docker-components-and-oci.html

[13] Kubernetes project, https://kubernetes.io

[14] Cloud Native Wiki by Aqua, https://www.aquasec.com/cloud-native-academy/

[15] Podman Red Hat Blog, https://developers.redhat.com/blog/2020/09/25/rootless-containers-with-podman-the-basics#why_podman_

[16] Podman project, https://podman.io/

[17] CRI-O project, https://cri-o.io/

[18] Sysdig 2021 Container security and usage report, https://dig.sysdig.com/c/pf-2021-container-security-and-usage-report?x=u_WFRi&utm_source=gated-organic

[19] Sysdig 2022 Cloud-Native Security and Usage Report, https://bit.ly/3CQ4CmD

[20] A brief overview of the Container Network Interface (CNI) in Kubernetes, https://www.redhat.com/sysadmin/cni-kubernetes

[21] Introducing CRI-O 1.0, https://www.redhat.com/en/blog/introducing-cri-o-10

[22] k3s project, https://k3s.io/

[23] k3s documentation, https://rancher.com/docs/k3s/latest/en/architecture/

[24] TCG Specification Architecture Overview, https://trustedcomputinggroup.org/wp-content/uploads/TCG_1_4_Architecture_Overview.pdf

[25] TCG Trusted Platform Module Library Part 1: Architecture, https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part1_Architecture_pub.pdf

[26] TCG PC Client Platform Firmware Profile Specification, https://trustedcomputinggroup.org/wp-content/uploads/TCG_PCClientSpecPlat_TPM_2p0_1p04_pub.pdf

[27] TCG DICE Attestation Architecture, https://trustedcomputinggroup.org/wp-content/uploads/DICE-Attestation-Architecture-r23-final.pdf

[28] Remote Attestation Procedures Archive, IETF, https://datatracker.ietf.org/doc/active/#id-rats

[29] S. C. Helble, I. D. Kretz, P. A. Loscocco, J. D. Ramsdell, P. D. Rowe, and P. Alexander, "Flexible mechanisms for remote attestation", ACM Trans. Priv. Secur., vol. 24, sep 2021, DOI 10.1145/3470535

[30] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not", 2015 IEEE Trustcom/BigDataSE/ISPA, 2015, pp. 57–64, DOI 10.1109/Trustcom.2015.357

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

[32] IMA Template Management Mechanism, https://docs.kernel.org/security/IMA-templates.html

[33] I. Pedone, D. Canavese, and A. Lioy, "Trusted computing technology and proposals for resolving cloud computing security problems", Cloud Computing Security: Foundations and Challenges (J. Vacca, ed.), pp. 373–386, CRC Press, 08 2020, DOI 10.1201/9780429055126-31

[34] Keylime Documentation, Release 6.4.2, July 2022, https://readthedocs.org/projects/keylime-docs/downloads/pdf/latest/

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

[36] M. De Benedictis and A. Lioy, "Integrity verification of docker containers for a lightweight cloud environment", Future Generation Computer Systems, vol. 97, 2019, pp. 236–246, DOI https://doi.org/10.1016/j.future.2019.02.026

Appendix A

User’s manual

This appendix provides the steps needed to configure and use a testbed composed of two nodes, one Attester (which will be the target of the remote attestation and runs the pods) and one Verifier.

A.1 Attester Machine

The first thing to do is to configure the host system which will be the attester machine in the RA process. As a prerequisite, a TPM 2.0 chip must be present.

A.1.1 Install k3s

After the installation of the OS, which is Ubuntu Server 20.04 LTS, install Lightweight Kubernetes:

```
curl -sfL https://get.k3s.io | sh -
```

Check if the node is ready through the command:

```
k3s kubectl get node
```

You should see:

```
NAME            STATUS   ROLES                  AGE   VERSION
torsec          Ready    control-plane,master   11s   v1.23.6+k3s1
```

If you need it, from the official website (https://k3s.io) you can retrieve the docs for a detailed installation.

A.1.2 Patching the Linux Kernel

Clone the git repository of the stable Linux Kernel:

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

After this operation, the directory linux-stable will be created. Select the latest stable release branch:

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

Then, to obtain the custom Linux kernel, apply the patches provided with the thesis source code:

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

$ git am --signoff < /patches_dir/0001-ima_cgn_template.patch
$ git am --signoff < /patches_dir/0002-ima_mns_template.patch
$ git am --signoff < /patches_dir/0003-entry_hash_256_bit.patch
$ git am --signoff < /patches_dir/0004-ima_dep_cgn_template.patch
$ git am --signoff < /patches_dir/0005-ima_cache_clp_patch.patch
$ git am --signoff < /patches_dir/0006-ima_cgpath_template.patch
```

A.1.3 Preliminary Steps

In order to proceed with the kernel compilation, you first need to perform some steps. The first one is to install the dependencies:

```
$ sudo apt-get update
$ sudo apt-get install git fakeroot build-essential ncurses-dev
  xz-utils libssl-dev bc flex libelf-dev bison dwarves
```

Copy the current kernel configuration file:

```
$ cp -v /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-cgpath
Default integrity hash algorithm  -> SHA256
Default template hash algorithm   -> SHA256
```

Save the modifications and exit. Now open the configuration file:

```
$ sudo nano .config
```

And in order to avoid compilation errors comment the following lines (in the section Certificates for signature checking):

```
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"
```

Now you are ready to compile the kernel and to make things faster you can run:

```
$ sudo make localmodconfig
```

which will skip unneeded drivers and will speed up the compilation process.

A.1.4 Kernel Compilation

The kernel compilation and installation can be done through the following commands:

```
$ sudo make -j 6
$ sudo make modules
$ sudo make modules_install
$ sudo make install
```

this operation will require some time. Reboot and add a custom IMA policy in a file called /etc/ima/ima-policy, for example:

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

After rebooting again you should be able to see the patched IMA module working, by looking in the file:

```
/sys/kernel/security/ima/ascii_runtime_measurements
```

A.1.5 Keylime Installation

First of all install the following dependencies:

```
$ sudo apt install libssl-dev swig python3-pip
```

The Keylime framework needs a version of `libtss2 >= 2.4.0`, and since Ubuntu 20.04 has by default the version 2.3.2, you need first to uninstall it:

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

Now you can manually build and install `libtss2 version >= 2.4.0`, but first, install the following dependencies:

```
$ sudo apt install autoconf autoconf-archive libglib2.0-dev libtool
  pkg-config libjson-c-dev libcurl4-gnutls-dev
```

Now you are ready to install all the tools required to manage the TPM 2.0 chip.

- Installation of libtss2

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

- Installation of tpm2-tools

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

- Installation of tpm2-abrmd

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

Configure TPM Command Transmission Interface (TCTI):

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

Start the Access Broker Resource Manager service:

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

And check if it is working:

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

You should read "active (running)". To check also if the tpm2 tools are working properly, you can run the command:

```
$ tpm2_pcrread
```

Which will show to you the content of the PCRs banks. Now you can finally install the framework, so move into the directory 'keylime' provided with the thesis source code, install the script and copy the configuration file:

```
$ cd keylime
$ sudo pip3 install . -r requirements.txt
$ sudo cp keylime.conf /etc/
```

A.1.6 Keylime Agent Configuration

To configure the agent on the attester machine, open the configuration file:

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

you can see that the file is divided into sections and each section has several parameters. For the Agent configuration, you need to modify only the [general] and the [cloud _agent] sections. In the [general] tag, find the receive_revocation_ip parameter and put the IP address of the attester machine, for example:

```
receive_revocation_ip = 192.168.0.100
```

In the [cloud _agent] tag set the cloudagent_ip which is the IP address of the attester machine, the registrar_ip which is the IP address of the registrar, and the agent_uuid, for example:

```
cloud_agent_ip = 192.168.0.100
registrar_ip = 192.168.0.114
agent_uuid = d432fbb3-d2f1-4a97-9ef7-75bd81c00000
```

Note that the agent_uuid shown in the example is the default one. Then set the hash algorithm:

```
tpm_hash_alg = sha256
```

A.1.7 Installation of the systemd service

Now you can install the keylime_agent as a systemd service. From the keylime directory move into /services and edit the file installer.sh in this way:

```
# prepare keylime service files and store them in systemd path

sed "s|KEYLIMEDIR|$KEYLIMEDIR|g" $BASEDIR/keylime_agent.service.template
    > /etc/systemd/system/keylime_agent.service

#sed "s|KEYLIMEDIR|$KEYLIMEDIR|g"
    $BASEDIR/keylime_registrar.service.template
    > /etc/systemd/system/keylime_registrar.service

#sed "s|KEYLIMEDIR|$KEYLIMEDIR|g"
    $BASEDIR/keylime_verifier.service.template
    > /etc/systemd/system/keylime_verifier.service


# set permissions

chmod 664 /etc/systemd/system/keylime_agent.service
#chmod 664 /etc/systemd/system/keylime_registrar.service
#chmod 664 /etc/systemd/system/keylime_verifier.service
chmod 666 /etc/systemd/system/keylime_agent_secure.mount

# enable at startup

systemctl enable keylime_agent.service
#systemctl enable keylime_registrar.service
#systemctl enable keylime_verifier.service
systemctl enable keylime_agent_secure.mount
```

Launch the following commands:

```
$ sudo groupadd keylime
$ sudo useradd keylime
```

And then run the script:

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

Finally, start the service and check if it is working:

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

You should see "active(running)".

A.2 Verifier Machine

Now we are going to configure the system which will be the verifier machine in the RA process. For simplicity, the components Verifier, Registrar, and Tenant of the Keylime Framework will be put all together in the same machine. In this case, a TPM 2.0 chip is not mandatory, it may be just emulated. The first thing to do is to install the OS, which again is Ubuntu Server 20.04 LTS. Then, follow the instructions for the Keylime installation indicated in section A.1.5.

A.2.1 Registrar

Open the configuration file:

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

and find the [registrar] tag. Then customize the registrar_ip by putting the IP address of the verifier machine, for example:

```
registrar_ip = 192.168.0.114
```

A.2.2 Verifier

Open the configuration file:

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

and find the [cloud_verifier] tag. Then customize the cloudverifier_ip, the registrar_ip, and the revocation_notifier_ip, by putting the IP address of the verifier machine, for example:

```
cloudverifier_ip = 192.168.0.114
registrar_ip = 192.168.0.114
revocation_notifier_ip = 192.168.0.114
```

A.2.3 Tenant

Open the configuration file:

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

and find the [tenant] tag. Then customize the cloudverifier_ip and the registrar_ip by putting the IP address of the verifier machine, for example:

```
cloudverifier_ip = 192.168.0.114
registrar_ip = 192.168.0.114
```

Then, from the attester machine retrieve the PCRs values with indexes 0-9 of the SHA256 bank, through the command:

```
$ sudo tpm2_pcrread
```

and copy them in the tpm_policy parameter in one line, as a JSON object, like:

```
tpm_policy = {"0": ["47D..."], "1":["25C..."], ..., "9":["4C3..."]}
```

Finally, go into the Keylime directory and copy the content of tpm_cert_store directory in /var/lib/keylime/tpm_cert_store/:

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

A.2.4 Installation as systemd services

Now you can install the keylime_registrar and the keylime_verifier as systemd services. From the keylime directory move into /services and edit the file installer.sh in this way:

```
# prepare keylime service files and store them in systemd path

#sed "s|KEYLIMEDIR|$KEYLIMEDIR|g"$BASEDIR/keylime_agent.service.template
    > /etc/systemd/system/keylime_agent.service

sed "s|KEYLIMEDIR|$KEYLIMEDIR|g"
    $BASEDIR/keylime_registrar.service.template
    > /etc/systemd/system/keylime_registrar.service

sed "s|KEYLIMEDIR|$KEYLIMEDIR|g"
    $BASEDIR/keylime_verifier.service.template
    > /etc/systemd/system/keylime_verifier.service


# set permissions
```

```
#chmod 664 /etc/systemd/system/keylime_agent.service
chmod 664 /etc/systemd/system/keylime_registrar.service
chmod 664 /etc/systemd/system/keylime_verifier.service
chmod 666 /etc/systemd/system/keylime_agent_secure.mount

# enable at startup

#systemctl enable keylime_agent.service
systemctl enable keylime_registrar.service
systemctl enable keylime_verifier.service
systemctl enable keylime_agent_secure.mount
```

Launch the following commands:

```
$ sudo groupadd keylime
$ sudo useradd keylime
```

And then run the script:

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

Finally, start the services and check if they are working:

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

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

You should see "active(running)" for both.

A.3 Starting RA process

In order to start the RA, you need to compute the whitelists related to the host system and pods. You can create the one related to the host system through the program whitelist_generator.cpp, while the ones related to pods are provided with the thesis source code in the folder pod_whitelists_. By way of information, only the whitelists of configuration pods and the whitelist associated with base image nginx are provided. In the Attester machine compile the program:

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

If you want to create the whitelist associated with the folder /usr/bin/ launch:

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

A file called "whitelist" will be generated. Now you have to create the pod list in the format pod UUID - whitelist path. You can retrieve the pod UUID list through the command:

```
kubectl get pods --all-namespaces -o

custom-columns=PodName:.metadata.name,PodUID:.metadata.uid
```

while the whitelist path corresponds to the pod_whitelist folder. Now you can send to the Verifier Machine the whitelist file just generated and the folder containing the pod list and one whitelist for each pod.

In the Verifier Machine, create also the exclude list associated with the host:

```
nano exclude_l
```

and put the content of the following regular expression:

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

which means "you have to exclude any file except for those present in /usr/bin/". You have also to create an empty file called "payload":

```
nano payload
```

which will be encrypted with the bootstrap key generated during the three-party bootstrap key derivation protocol.

Now you are ready to launch the command:

```
$ sudo keylime_tenant -c add -u <UUID> -t <IP>
-f payload --allowlist whitelist --exclude exclude_l
--pod_list pods_list
```

You should see the message:

```
" QUOTE from <IP > validated "
```

which indicates that the RA started. From now on you can use the Keylime CLI explained in the next section to communicate with the framework and monitor the attestation results.

A.4 Keylime CLI

This section provides the commands offered by the Keylime CLI to interact with the framework. Each command respects the format:

```
    keylime_tenant -c [command]
```

where the -c option can take one of the following keywords.

A.4.1 Status

The Status command is the one that consents to retrieve the status of a registered agent. Of course, the registration has to be done both at the Registrar and the Verifier machines. An example of this command is:

```
 $ sudo keylime_tenant -c status -u [UUID]
```

where the placeholder [UUID] represents the agent UUID. If not specified, the default UUID is used, which corresponds to d432fbb3-d2f1-4a97-9ef7-75bd81c00000. Optionally, if more than one Verifier is used, can be specified also the specific verifier IP address in which the agent is registered, through the option:

```
$ sudo keylime_tenant -c status -u [UUID] -v [Verifier_IP]
```

A.4.2 Add

The Add command is the one used to register an agent at the Verifier. It enables the periodic remote attestation and can take the following parameters:

- -u [UUID].

Placeholder [UUID] is the agent UUID to be added. If not specified, the default UUID is used, which corresponds to d432fbb3-d2f1-4a97-9ef7-75bd81c00000.

- -v [Verifier_IP].

Placeholder [Verifier_IP] is the IP address of the Verifier machine where the agent has to be registered. If not specified, it is used the IP address inserted into:

/etc/keylime.conf.

- -t [Agent_IP].

Placeholder [Agent_IP] is the IP address of the agent to be added.

- -f [payload].

Placeholder [payload] is a file to be encrypted with the bootstrap key.

- –exclude [exclude_list].

Placeholder [exlude_list] is the file containing a regular expression with the files to be excluded in the validation process of the ML.

- –allowlist [whitelist].

Placeholder [whitelist] is the file containing the golden values used to validate ML entries.

- –pods_list [pod_list].

Placeholder [pod_list] is a file containing the pod UUIDs to be considered allowed in any Attester machine. The entry format follows the schema: podUUID /whitelist_path.

An example of this command is:

```
$ sudo keylime_tenant -c add -u 90e71d86-13e0-4bd3-9ec4-1521f10a5194
-t 192.168.0.103 -f payload --exclude exclude_host
--allowlist whitelist2 --pods_list pods_list2
```

A.4.3 Update

The Update command is the one used to update an already registered agent. It follows the same logic as the add command, so it uses the same parameters. An example of this command is:

```
$ sudo keylime_tenant -c update -u
  d432fbb3-d2f1-4a97-9ef7-75bd81c00000
  -t 192.168.0.100 -f payload --exclude exclude_host
  --allowlist whitelist --pods_list pods_list
```

A.4.4 Delete

The Delete command is the one used to remove a registered Agent from the Verifier. It can take parameters:

- -u [UUID].

Placeholder [UUID] is the agent UUID to be removed. If not specified, the default UUID is used, which corresponds to d432fbb3-d2f1-4a97-9ef7-75bd81c00000.

- -v [Verifier_IP].

Placeholder [Verifier_IP] is the IP address of the Verifier machine where the agent has to be deleted. If not specified, it is used the IP address inserted into:

/etc/keylime.conf. An example of this command is:

```
$ sudo keylime_tenant -c delete -u
  d432fbb3-d2f1-4a97-9ef7-75bd81c00000
  -t 192.168.0.100
```

A.4.5 Pods

Pods is a new command added during the thesis work. It can take the following parameters:

- -u [UUID].

Placeholder [UUID] is the agent UUID in which the pods to be checked are running. If not specified, the default UUID is used, which corresponds to d432fbb3-d2f1-4a97-9ef7-75bd81c00000.

- -v [Verifier_IP].

Placeholder [Verifier_IP] is the IP address of the Verifier machine where the agent has been registered. If not specified, it is used the IP address inserted into:

/etc/keylime.conf.

- –pod_id [POD_UUID]

Placeholder [POD_UUID] represents one of the registered pods for the specified agent.

- –allowlist [path/new_pod_whitelist]

Placeholder [path/new_pod_whitelist] represents the path of a new whitelist related to the pod specified via –pod _id.

- –exclude [path/new_pod_exclude_list]

Placeholder [path/new_pod_exclude_list] represents the path of a new exclude list of the pod specified via –pod_id.

Examples

Some examples of this command.

```
$ sudo keylime_tenant -c pods
```

it shows the list of registered pods for default agent d432fbb3-d2f1-4a97-9ef7-75bd81c00000.

```
$ sudo keylime_tenant -c pods --pod_id
  226aed86-763b-4a3e-925b-82e50146171e
```

It shows:

- the operational state (among START, TRUSTED, UNTRUSTED)
- the allowlist
- the exclude list

for default agent d432fbb3-d2f1-4a97-9ef7-75bd81c00000 and pod 226aed86-763b-4a3e-925b-82e50146171e.

```
keylime_tenant -c pods --pod_id
                226aed86-763b-4a3e-925b-82e50146171e
               --allowlist /home/torsec/pod_whitelists/nginx
```

it consents to update the allowlist related to default agent d432fbb3-d2f1-4a97-9ef7-75bd81c00000 and pod 226aed86-763b-4a3e-925b-82e50146171e.

A.4.6 Reglist

The command Reglist is the one used to retrieve the list of agents registered at the Registrar component. It does not take parameters. An example of this command is:

```
 $ sudo keylime_tenant -c reglist
```

A.4.7 Regdelete

The command Regdelete is the one used to delete a registered agent from the Registrar. It takes as parameter:

- -u [UUID].

Placeholder [UUID] is the agent UUID to be removed from the Registrar. If not specified, the default UUID is used, which corresponds to d432fbb3-d2f1-4a97-9ef7-75bd81c00000.

An example of this command is:

```
$ sudo keylime_tenant -c regdelete -u
  90e71d86-13e0-4bd3-9ec4-1521f10a5194
```

A.4.8 Cvlist

The command Cvlist is the one used to retrieve the list of agents currently registered at the Verifier component. If more than one Verifier is used, the verifier IP address to be checked can be specified through the option:

- -v [Verifier_IP].

If not specified, it is used the verifier IP address inserted into /etc/keylime.conf. An example of this command is:

```
$ keylime_tenant -c cvlist -v 192.168.0.114
```

A.4.9 Reactivate

The command Reactivate can be used only when an agent has been terminated or has failed its periodic attestation, otherwise, it is considered a forbidden operation. It uses parameters:

- -u [UUID].

Placeholder [UUID] is the agent UUID to be reactivated. If not specified, the default UUID is used, which corresponds to d432fbb3-d2f1-4a97-9ef7-75bd81c00000.

- -v [Verifier_IP].

Placeholder [Verifier_IP] is the IP address of the Verifier machine where the agent has been registered. If not specified, it is used the IP address inserted into:

/etc/keylime.conf. An example of this command is:

```
$ sudo keylime_tenant -c reactivate -u
  90e71d86-13e0-4bd3-9ec4-1521f10a5194
```

A.5 Keylime Web app

As an alternative to the CLI, the Keylime framework offers also a Web app that works with REST APIs.

A.5.1 Installation

Move into the Verifier Machine and open the Keylime Configuration file located in /etc/keylime.conf. The section [Tenant] has been already configured as explained in section A.2.3, while in the section [Webapp] set the following parameters:

```
    webapp_ip = <VERIFIER_IP>
    webapp_port = 444
```

the port needs to be set to 444 because 443 is the one chosen by the k3s server (if you want to set up a cluster, as explained in the following section). Now you can run:

```
keylime_webapp
```

to launch the Web app which will be reachable at the URI:

```
https://<webapp_ip>:<webapp_port>/webapp/
```

A.6 Cluster Set Up

If you want to turn the testbed into a k3s cluster, you have to move into the Verifier Machine and install k3s with the following command:

```
curl -sfL https://get.k3s.io | INSTALL_K3S_EXEC='server
--cluster-init
--write-kubeconfig-mode=644' sh -s
```

The Verifier Machine will run as Master Node. Then, move in the Attester Machine and first, you have to install the Kubectl component:

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/
kubectl"
```

Download the kubectl checksum file:

```
curl -LO "https://dl.k8s.io/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/
kubectl.sha256"
```

And validate it:

```
echo "$(cat kubectl.sha256)  kubectl" | sha256sum --check
```

If valid, you should see a message like this:

```
kubectl: OK
```

Install it:

```
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

and check if the configuration is good by looking at the cluster state through the command:

```
kubectl cluster-info
```

You should see a URL response. Then retrieve the token from:

```
# cat /var/lib/rancher/k3s/server/node-token
```

copy and place it in the placeholder `<TOKEN>` in the commands below, while the `<VERIFIER_IP>` is the IP address of the Verifier machine:

```
curl -sfL https://get.k3s.io | K3S_URL=
https://<VERIFIER_IP>:6443 K3S_TOKEN=${<TOKEN>}
K3S_KUBECONFIG_MODE="644" sh -

sudo k3s agent --server https://<VERIFIER_IP>:6443
--token ${<TOKEN>::server:9200aee61eb292ee60cfdb6cd25cfc02}
```

Now from the Verifier move in:

```
/etc/rancher/k3s/k3s.yaml
```

and change the server IP with the Verifier IP address. Now copy from the Verifier to the Attester the content of:

```
cat /etc/rancher/k3s/k3s.yaml
```

inside:

```
/root/.kube/config
```

Now, from the Verifier run:

```
k3s kubectl get node -o wide
```

You will see two machines, one with ROLE "control-plane, etcd, master" corresponding to the Verifier machine, and the other with ROLE "`<none>`" corresponding to the Attester machine. In the field KERNEL-VERSION, you will see that the Attester machine has the patched kernel version. Now you have successfully created a cluster with a Master Node running the Keylime Verifier, Tenant and Registrar components, and a Worker Node running the Keylime Agent.

A.6.1 Add a Deployment

To add a deployment meant to run the specified pods in the Worker you have to add a *label*. So, from the Attester machine run the command:

```
kubectl label nodes torsec-k3s-nuc-01 disktype=ssd
```

you should see an output like this:

```
node/torsec-k3s-nuc-01 labeled
```

which means that the node has been successfully labeled. Now save into a file called nginx-deployment.yaml the following manifest:

```
    apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
      nodeSelector:
          disktype: ssd
```

and then run the command:

```
kubectl apply -f nginx-deployment.yml
```

Now to check if everything is good run:

```
kubectl get pods -l app=nginx --output=wide
```

you should see that 5 nginx pods are correctly running in the Worker node (i.e. Attester node). If you want to add new deployments and retrieve the pod UUID of all the pods currently running in the Worker node, you can run the command:

```
kubectl get pods  -o custom-columns=PodName:.metadata.name,
PodUID:.metadata.uid
```

the output, since there is only one worker node, will display exactly the pod UUID list to be given to the verifier.

A.6.2 Test The Solution

Now you can check if the Remote Attestation is working properly.

One pod untrusted in a trusted attester system

Login to any of the pods with the command:

```
kubectl exec <pod_name> -n <pod_namespace> -it -- bash
```

if the related /bin/bash/ measured was not present in the associated whitelist, the pod will be immediately found untrusted and the file /bin/bash put within the "files not found" list. The Keylime Verifier will find out that the chosen pod has been compromised while the Attester system and the other pods stay trusted. As an alternative to obtain the same result, it is possible to change by hand the pod-associated whitelist, either deleting a file or changing its measure. It is also possible to test the solution by adding a measure within the pod whitelist, associated with a simple script like:

```
#!/bin/bash
echo "Hello World"
```

The measure can be retrieved after the execution from the file:

```
/sys/kernel/security/ima/ascii_runtime_measurements
```

and put in the format: *measure /bin/bash/wrong_measure*. Login to the pod to be tested and run:

```
cd /bin/bash
echo '#!/bin/bash echo' > wrong_hash
chmod +x wrong_hash
./wrong_hash
```

The Keylime Verifier will find out that the chosen pod has been compromised while the Attester system and the other pods stay trusted. The file wrong_hash will be put in the list "file hash errors".

Untrusted attester system

It is possible to check also that the integrity of the host system has been evaluated correctly. Since it is the only folder taken into consideration, move into:

```
$ cd /usr/bin
```

and create a script:

```
$ nano hello.sh
```

with the content:

```
#!/bin/bash
echo "Hello World"
```

Change the executable permission:

```
chmod +x hello.sh
```

and run the script:

```
./hello.sh
```

Now you will see that the operational state passes from "Get Quote" to "Invalid Quote" and the periodic remote attestation terminates. This happens because the file "hello.sh" does not belong to the whitelist associated with the host. As an alternative, you can also modify the pod_list associated with the host, by deleting a valid pod UUID that is actually running on the host. In this case, the host system will be evaluated as untrusted because in the Measurement Log there will be entries associated with the deleted pod, which will be considered as an unknown one. To obtain the latter behavior it is also possible to create a new deployment:

```
nano deployment.yaml
```

with content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: alpine-deployment
spec:
  selector:
    matchLabels:
      app: alpine
  replicas: 1
  template:
    metadata:
      labels:
        app: alpine
    spec:
      containers:
      - name: alpine
        image: alpine
        ports:
        - containerPort: 10000
      nodeSelector:
        disktype: ssd
```

and run:

```
kubectl apply -f deployment.yaml
```

Now there will be an unregistered alpine pod which causes the system to be considered untrusted (i.e. with operational state set to INVALID QUOTE).

To delete the new alpine deployment you can run:

```
kubectl delete deployment alpine-deployment
```

A.7 Keylime REST APIs - CLI

This section provides the Keylime REST APIs modified and added during the thesis work. The existing APIs affected by modifications are presented in the first paragraph, while the new ones are in the second. You can find the related code in the "Keylime/keylime/" folder provided in the thesis source code, specifically within files:

- *tenant.py*, for the Kelylime CLI
- *tenant_webapp.py*, for the Keylime Webapp

A.7.1 Existing REST APIs

`GET /v2/agents/{agent_id:UUID}`

This GET API is in charge to provide the *status* operation for an agent already registered at the verifier. The modification is concerned with showing also the pods' status, specifically for each one of the pods is present:

- The operational state, which could be 0 (START), 1 (TRUSTED), or 2 (UNTRUSTED).
- The whitelist.
- The exclude list.
- The list of files present in the ML but not found in the whitelist.
- The list of files present in the ML, in the whitelist, but with a wrong measure.

The response JSON object is composed of the following fields, where the field **pods** is the new one:

- **operational_state**, that corresponds to the current state of the requested Keylime Agent.
- **v**, that is the V part of the bootstrap key.

- **ip**, that is the IP address of the Keylime Agent
- **port**, that is the port used by the Keylime Agent to connect to the Keylime Verifier.
- **tpm_policy**, that is used to specify the PCRs that must be included in each TPM quote contained in the IR.
- **vtpm_policy**, that is used as the tpm_policy when virtual PCRs are involved.
- **meta_data**, that specifies the Keylime Agent-related metadata.
- **allowlist_len**, that corresponds to the number of lines in the whitelist associated with the physical system.
- **mb_refstate_len**, that corresponds to the length of the policy associated with the measured boot.
- **accept_tpm_hash_algs**, that specifies the list of the hash algorithms accepted by the TPM.
- **accept_tpm_encryption_algs**, that specifies the list of the encryption algorithms accepted by the TPM.
- **accept_tpm_signing_algs**, that specifies the list of the signing algorithms accepted by the TPM.
- **hash_alg**, that is the actual hash algorithm used by the TPM.
- **enc_alg**, that is the actual encryption algorithm used by the TPM.
- **sign_alg**, that is the actual signing algorithm used by the TPM.
- **verifier_id**, that is a unique identifier for each verifier instances.
- **verifier_ip**, that is the IP address of the Keylime Verifier server binds to.
- **verifier_port**, that is the port of the Keylime Verifier server binds to.
- **severity_level**, that in case of a failure corresponds to the severity of the failure.
- **last_event_id**, which is the identifier of the last failure event with maximum severity.
- **pods**, that is a JSON object that specifies the pods registered in the Keylime Agent. Each pod is described through a JSON object containing the fields specified above.

`POST /v2/agents/{agent_id:UUID}`

This POST API is in charge to provide the *add* operation for registering an agent at the verifier. The modification is concerned with providing in the request the additional data needed for the proposed solution, which are the pod list and whitelists.

The request JSON object is composed of the following fields, where the field **pods** is the new one:

- **v**, that is the V part of the bootstrap key.
- **cloudagent_ip**, that is the IP address of the Keylime Agent for the Keylime Verifier.
- **cloudagent_port**, that is the port used by the Keylime Agent to connect to the Keylime Verifier.
- **tpm_policy**, that is used to specify the PCRs that must be included in each TPM quote contained in the IR.
- **vtpm_policy**, that is used as the tpm_policy when virtual PCRs are involved.
- **meta_data**, that specifies the Keylime Agent-related metadata.
- **allowlist**, that corresponds to the whitelist associated with the physical system.
- **mb_refstate**, that corresponds to the policy associated with the measured boot.
- **ima_sign_verification_keys**, that corresponds to the list of IMA public keys for signature verification.
- **revocation_key**, which corresponds to the RSA private key to be used by the Keylime Verifier to sign a revocation message for this Keylime Agent.
- **accept_tpm_hash_algs**, that specifies the list of the hash algorithms accepted by the TPM.
- **accept_tpm_encryption_algs**, that specifies the list of the encryption algorithms accepted by the TPM.
- **accept_tpm_signing_algs**, that specifies the list of the signing algorithms accepted by the TPM.
- **pods**, that is a JSON object that specifies the pods to be registered in the Keylime Agent. Each pod is specified through its UUID and a JSON object that contains the allowlist and the exclude list.

A.7.2 New REST APIs

The new REST APIs added to the framework with the thesis work are presented in this section.

`GET /v2/agents/{agent_id:UUID}/pods`

This GET API is in charge to show the pod UUIDs list of the registered pods for the Keylime Agent specified through the {agent_id:UUID} placeholder. The response JSON object will contain:

- the UUID of the Keylime Agent
- the list of the pod UUIDs registered for that Keylime Agent

An example of the response could be:

```
{
   code: 200,
   status: "OK",
   results: {
                  uuid: "d432fbb3-d2f1-4a97-9ef7-75bd81c00000",
                  pod_ids:
                    0: "226aed86-763b-4a3e-925b-82e50146171e",
                    1: "bead1494-a2ff-4b2b-bead-97f911a0039f",
                    ...
            }
 }
```

`POST /v2/agents/agent_id:UUID/pods`

This POST API is in charge to add at the Keylime Verifier the pod UUIDs list associated with the Keylime Agent specified through the {agent_id:UUID} placeholder. The pod list is a dictionary of pod UUIDs, each one associated with its whitelist and (optionally) exclude list.

The Request JSON object will contain a dictionary of pod identifiers, each one associated with a JSON object containing the fields:

- **allowlist**
- **exclude list**

`PUT /v2/agents/{agent_id:UUID}/pods`

This PUT API is in charge to modify at the Keylime Verifier the pod UUIDs list associated with the Keylime Agent specified through the {agent_id:UUID} placeholder.

The Request JSON object will contain a dictionary of pod identifiers, each one associated with a JSON object containing the fields:

- **allowlist**
- **exclude list**

`GET /v2/agents/{agent_id:UUID}/pods/{pod_id}`

This GET API is in charge to show the status of a registered pod specified through the {pod_id} placeholder. Specifically, in the response JSON object, there will be:

- the pod UUID;
- the pod operational state;
- the pod whitelist;
- the pod exclude list (if present).
- The list of files present in the ML but not found in the whitelist.
- The list of files present in the ML, in the whitelist, but with a wrong measure.

`PUT /v2/agents/{agent_id:UUID}/pods/{pod_id}`

This PUT API is in charge to modify the whitelist or the exclude list of the pod specified through the {pod_id} placeholder.

The Request JSON object will contain:

- the **allowlist** of the {pod_id}
- the **exclude list** of the {pod_id}

`DELETE /v2/agents/{agent_id:UUID}/pods/{pod_id}`

This DELETE API is in charge to remove the pod specified through {pod_id} from the Keylime Agent identified through {agent_id:UUID}.

`GET /v2/agents/{agent_id:UUID}/pods/{pod_id}/allowlist`

This GET API is in charge to show the whitelist associated with the pod specified through the {pod_id} placeholder.

`PUT /v2/agents/{agent_id:UUID}/pods/{pod_id}/allowlist`

This PUT API is in charge to substitute the whitelist associated with the pod specified through the {pod_id} placeholder, with the new whitelist provided in the request body.

The Request JSON object will contain the new whitelist for the {pod_id}

`GET /v2/agents/{agent_id:UUID}/pods/{pod_id}/exclude`

This GET API is in charge to show the exclude list associated with the pod specified through the {pod_id} placeholder.

`PUT /v2/agents/{agent_id:UUID}/pods/{pod_id}/exclude`

This PUT API is in charge to substitute the exclude list associated with the pod specified with the {pod_id} placeholder, with the new exclude list provided in the request body.

A.8 Keylime REST APIs - Tenant Webapp

`GET /v2/agents/{agent_id:UUID}`

The response JSON object of this API has the same format as the API `GET /v2/agents/{agent_id:UUID}`, with an additional field id representing the agent UUID.

`POST /v2/agents/{agent_id:UUID}`

This POST API is in charge to register a Keylime Agent at the Keylime Verifier, providing an encrypted payload to the agent UUID. The request JSON object has the following fields:

- **agent_ip**, that is the Keylime Agent IP address.
- **ptype**, that is the payload type. It can assume values 0 (FILE), 1 (KEY-FILE), or 2(CA_DIR).
- **file_data**, that is the payload. If the field ptype is equal to 0, then it is encrypted by the Keylime Tenant with a random bootstrap key. If the field ptype is equal to 1, then it is encrypted with the bootstrap key specified within keyfile_data.
- **keyfile_data**, this field contains a bootstrap key which is used when the field ptype is equal to 1.
- **include_dir_data**, this field is needed when the field ptype is equal to 2, and it contains a list of data to be sent in a zip to the Keylime Agent.
- **include_dir_name**, this field is needed when the field ptype is equal to 2, and it contains the names of file data specified in the include_dir_data.
- **ca_dir**, this field is needed when the field ptype is equal to 2, and it contains the path of the directory on which the Keylime Tenant is running.
- **ca_dir_pw**, this field is needed when the field ptype is equal to 2, and it contains the CA password.
- **tpm_policy**, that is used to specify the PCRs that must be included in each TPM quote contained in the IR.
- **vtpm_policy**, that is used as the tpm_policy when virtual PCRs are involved.
- **a_list_data**, that is the whitelist of the host system.
- **e_list_data**, that is the exclude list of the host system.
- **ima_sign_verification_keys**, that corresponds to the list of IMA public keys for signature verification.
- **mb_refstate**, that is the policy associated with the measured boot.
- **pods**, that represents a JSON object for each pod to be registered in the Keylime Agent. The JSON object contains the a_list_data for the whitelist, and the e_list_data for the exclude list.

All the other APIs are not reported because they are internally invoking the ones exposed by the CLI.

Appendix B

Developer's manual

B.1 IMA patch

This section provides the steps needed to create the IMA patch used in the proposed work. By way of information, the starting point has been implemented by the TORSEC research group of Polytechnic of Turin.

B.1.1 ima-cgpath template implementation

As explained in chapters 5 and 6, the ima-cgpath template has an additional field for the control group path that is unsupported by IMA, so the first thing to do is to move into the Linux kernel source code directory and open:

```
$ nano ./security/integrity/ima/ima_template.c
```

Then add to the array called **supported_fields** the following content:

```
      static const struct ima_template_field supported_fields[] = {
        ...
          {.field_id = "cg-path", .field_init =
              ima_eventcg_path_init,
           .field_show = ima_show_template_string},
    };
```

where:

1. "cg-path" represents the field identifier
2. "ima_eventcg_path_init" represents the function used to initialize the field value for the Measurement Events
3. "ima_show_template_string" represents the function used to write the field value in the Measurement Logs

Then in the same file define a new builtin template named "ima-cgpath" with format string "dep—cg-path—d-ng—n-ng":

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

the "dep" field represents the dependencies of the process that created the entry, and it has been defined by TORSEC research group of Polytechnic of Turin since also this parameter was not among those supported by IMA. The field "cg-path" is the control group path (needed to establish the pod UID), while "d-ng" and "n-ng" are respectively the file digest and the file path.

The next operations to do are to put the prototype of the function ima_eventcg_path_init within the file **ima_template_lib.h**:

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

and to put its definition in the file **ima_template_lib.c**:

```
 /*
* ima_eventcg_path_init - include the current task's cgroup
   path as part of the
* template data
*/

int ima_eventcg_path_init(struct ima_event_data *event_data,
                      struct ima_field_data *field_data)
{
char *cgroup_path_str = NULL;
struct cgroup *cgroup = NULL;
int rc = 0;
cgroup_path_str = kmalloc(PATH_MAX, GFP_KERNEL);
if (!cgroup_path_str)
              return -ENOMEM;
cgroup = task_cgroup(current, 1);
if (!cgroup)
              goto out;
rc = cgroup_path(cgroup, cgroup_path_str, PATH_MAX);
if (!rc)
              goto out;
       rc = ima_write_template_field_data(cgroup_path_str,
          strlen(cgroup_path_str), DATA_FMT_STRING, field_data);
       kfree(cgroup_path_str);
       return rc;
out:
       return ima_write_template_field_data("-", 1,
          DATA_FMT_STRING, field_data);
}
```

The function **ima_eventcg_path_init** uses three local variables:

- **cgroup_path_str** (line 9) which will represent the buffer for the control group path,
- **cgroup** (line 10) which will represent the cgroup data structure,
- **rc** (line 11) which will represent the return value.

The function **kmalloc()** is invoked to reserve the needed memory to contain the cgroup path, having as parameters:

- **PATH_MAX**, that is a define that specifies the bytes to be allocated, and you can find it in the file /include/uapi/linux/limits.h:

  ```
  #define PATH_MAX 4096 /* # chars in a path name including nul */
  ```

- **GFP_KERNEL** needed for kernel allocation.

Then, if the allocation is successful, the function **task_cgroup** will get the control group associated to hierarchy ID = 1 for the current Linux task specified in **current**. A this point, the function **cgroup_path** will set the path of the cgroup which was earlier put into the variable **cgroup_path_path_str**. Now the function **ima_write_template_field_data** will be invoked to write the template value within the **field_data** parameter, with data format **DATA_FMT_STRING**. The definition of **ima_write_template_field_data** can be found inside the file **ima_template_lib.c**.

Finally, open the **Kconfig** file and add the highlighted content, which consents to add the ima-cgpath template. In this way will be possible to use it as the default IMA template before proceeding with the kernel compilation, through the **make menuconfig** command, as explained in A.1.3:

```
   choice
        prompt "Default template"
        default IMA_NG_TEMPLATE
        depends on IMA
    ...
        config IMA_DEP_CGN_TEMPLATE
                bool "ima-dep-cgn"
config IMA_CGPATH_TEMPLATE
bool "ima-cgpath"
   endchoice

   config IMA_DEFAULT_TEMPLATE
                string
                depends on IMA
                default "ima" if IMA_TEMPLATE
                ...
                default "ima-dep-cgn" if IMA_DEP_CGN_TEMPLATE
default "ima-cgpath" if IMA_CGPATH_TEMPLATE
```