POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master Thesis

Remote Attestation on light VM

Supervisor
prof. Antonio Lioy

Candidate
Fabio Vallone

Accademic Year 2016-2017

To my parents, who helped me during this journey

Summary

In the last decade Cloud Computing has massively entered the IT world, changing the way services are offered to the final users. One important aspect of Cloud Computing is the needs to provide greater scalability and flexibility and this can be done by using a technique called Virtualization. This enables us to execute different virtualized ambient on a single piece of hardware. Each virtualized ambient can be seen as a node that offers some services to the final users by exchanging information with other nodes. For this reason the ability to trust each other is growing exponentially. This process is called Remote Attestation and it is from this consideration that the thesis work start.

When we say that a node is trustworthy, we are saying that we are sure that all the file that are loaded into memory are the ones that are supposed to be loaded. In other word we need to be sure that the code, and in general the files loaded on that node, have not been tampered by anyone in order to produce a different behaviour of the node. It is important to note that we are not checking in anyway that the behaviour of the machine is logically correct, but we are simply checking that no one has modified it, so bug and logical error can still occurs in the code. In order to prove its trustworthiness, a single node must be able to report his integrity status to a third one. This can be done by using the techniques and technologies developed by the *Trusted Computing Group* (hereafter TCG). The set of the solutions proposed by the TCG constitute the *Trusted Computing technology*. They are all based on a particular chip called *Trusted Platform Module* (TPM), which is now broadly available in most modern motherboard. Basically the TPM offers two type of operation: Secure data storage and cryptography. In particular the data can be stored in a very reduced set of registers, called PCRs. For the purpose of the Remote Attestation process, it is possible to store every measure done by the TPM inside the PCRs by simply extending the old value of the register with the new one by using an hash function.

All the techniques developed by the TCG were initially developed in order to work with physical machine. Since nowadays most of the node in the cloud are no longer physical machine, all the concept introduced with the Trusted Computing methodologies have been recently extended to cover also virtual nodes. However in the last years a new virtualization technique, called Light Virtualization, is raising fast. The main advantage of this new technique is that is generally faster and less resource consuming on the host machine. This is done by avoiding to load a full copy of the OS for each virtualized ambient, but instead, using layers and granting direct access to the resource of the machine to the various virtual nodes. In particular, in a standard virtualization mechanism, all the nodes are managed by a broker, called

hypervisor, while with the light Virtualization, they are managed directly by the host operating system. All the thesis work is focused on a particular light virtualization engine called *Docker*.

Less isolation means also less security. For that reasons we need to extended the work done by the TCG also to the light virtualization world. Some work in that direction has already been done by the TORSEC research group inside the Politecnico di Torino. They proposed a solution that makes use of a modified version of the *Open Attestation Framework* (OAT) and the IMA modules, in order to create an architecture that is able to attestate an host running on docker. This solution exploit the IMA module of the kernel in order to create a list of measurements associated with each container running inside an host. By exploiting the OAT framework it is possible for an Appraiser, to send an *Attestation Request* to a node, that respond with an *Integrity Report* (compiled using the list of measurement done by IMA). The IR is than checked by controlling that all the hashes are available also in a whitelist db. If some of the hashes into the IR are not contained into the whitelist db, then the container is probably under attack.

However the solution is based on a functionality available inside the docker storage driver *DeviceMapper* that has been extensively changed starting from v1.10. The thesis work starts by adapting the existing architecture to the latest version of docker. This is done by studying how DeviceMapper is changed after v1.10 and how it maps each container to the underlying devicemapper folder. After a brief analysis, a new logical mapping procedure has been developed. Though the solution is working, it also introduces a big overhead in terms of time when creating an Integrity Report. This big overhead comes from the extensive use of I/O operations in order to retrieve all the information necessary for the mapping. Since a big part of them are already available in the docker CLI, and the GO programming language is highly optimized for managing the Unix sys call, all the mapping logic is moved to a new docker CLI command, called *raInfo*.

After resolving thie initial issue, the work moves on by identifying some flaws of the architecture proposed and how it is possible to resolve them. In particular it was necessary to study how IMA works in detail, since, by the default, a particular file is measured only the first time it is actually loaded into memory. Since with docker is possible to share a portion of the host filesystem between multiple container, it is also possible that a particular file can be loaded multiple time inside different containers. However this file can be allowed only in some container and not in all of them, but since IMA measure that file only the first time, then we are not aware that this file is in execution on multiple containers. In order to resolve that problem, the IMA module has been changed by adding two kernel boot parameters that enables the deactivation of his internal caching mechanism.

In conclusion, the proposed solution by this thesis enables the Remote Attestation process on Docker v1.10+, and resolves some issues found during the previous development of the project. It also extends the IMA module by enabling to control each cache system independently. The solution is then compared to similar solutions available in the market.

Acknowledgements

A special thanks goes to *Tao Su*, PhD candidate inside the research group TORSEC of the Politecnico di Torino, who helped me during the development of the thesis work, and to *Marco De Benedictis*, researcher inside the TORSEC group of the Politecnico di Torino, who helped me during the writing of the thesis. A general thanks goes to all the members of the TORSEC research group, which were always available to help me when needed.

Contents

Chapter 1

Introduction

In the last years we have assisted to the rapid grow of *Cloud Computing*. It has changed the way IT services are offered and accessed from the users. Cloud Computing can be seen as an agglomerate of physical server who work together in order to provide greater flexibility, cost reduction and service availability. The pool of hardware resources assigned to a particular service can be changed to the need. With this infrastructure the user no longer needs to install a specific software in order to access its services and also the services can be accessed anywhere, the only requirements are an internet connection and a browser. However, with this new technology, we need to put special attention to the problem of IT security since it provides less control on the client connecting to the service.

With the grow of Cloud Computing we have seen also the grow of Virtualization Technique that enable us to emulates physical hardware resources in order to easily escalate our infrastructure when needed. This will lead us to the concept of Cloud Computing of today and, as defined by the *National Institute of Standards and Technology* (NIST): [1]

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

Some key features which a Cloud Computing has to offer are:

- **On-demand self-service**: It is possible to acquire, when needed, all the computation power necessary for the service to work, without the need of an interaction with the service provider
- **Broad network access**: Services are available into the network and can be accesses with standard mechanism and with different platform
- **Resource pooling**: Hardware resources are no longer dedicated to a single user or service, instead they are shared with multiple services and they can be dynamically reassigned at need

- **Rapid elasticity**: Resources can be dynamically acquired and released when needed in order to rapidly escalate to the service demands. Often resources are seen as unbounded to the service consumer
- **Measured service**: Hardware resources consumption can be monitored so that both the service provider and the service user are aware of it

Cloud Computing was developed in order to work with physical hardware, so that all the services runs directly on the Operating System of the server. With the grow of Virtualization technique, services provider, start to understand the benefit of using such technology. It enables them to run different services and application on the same physical hardware granting, at the same time, all the isolation level needed for security purposes. Also they are able to provide great scalability and fast response to the service user needs. All this technology is based on the figure of the Hypervisor, a special software controller that emulates and manages all the hardware resources needed to the various virtual machines. However one aspect that needs to be considered with the classical virtualization technique is the high overhead introduced especially for the management operations of a virtual machine, such as the startup process.

For resolving this issue, in the last years, a new virtualization technique is growing fast. This technique is called *Light Virtualization* [2] because it provides more or less the same features provided by a classical virtual machine but without the big overhead introduced by it. In fact the hypervisor is no longer used, instead the hardware resources are managed directly by the kernel of the host Operating System. In a light virtualized ambient a single Virtual Machine is called *Container*. In order to reduce the overhead, more containers share the same underlying virtualized Operating System so that it is not necessary to load all the virtual hardware resources and OSs for each containers. The files of the virtual OS are shared with a particular filesystem called *layer fs* (based on the linux Union FileSystem). Each container is isolated from other by using the kernel capabilities of the underlying host kernel.

In this context, IT Security, is assuming a very important role, no matter the technique that is used. This is due to the fact that a machine, physical or virtual, is always accessible by the external network and it is exposed to various type of attacks. In particular it is possible to manipulate a node in order to produce different behaviour. For this reason it is very important to have the ability to test the integrity status of a remote node in order to trust him. All the technology developed in that direction goes under the name of *Trusted Computing*.

This technology is developed by the *Trusted Computing Group* (TCG) [7] and have at the core the ability to perform the *Remote Attestation* process. At the beginning all the processes were developed in order to work with physical nodes. However, since most modern cloud rely on virtualization technique, they were extended in order to work also with this type of infrastructure. In particular it is possible to Remotely Attestate a classical Virtual Machine (so the one with the hypervisor) by using a component called *Integrity Verification Proxy* (IVP) [11] by simply putting trust in the host operating system and the IVP with the standard Remote Attestation procedure and then ask the IVP to instantiate a connection

with a VM if and only if the machine meets the requirements given at connection time by the client. However in order to resolve the bottleneck derived from the utilization of the standard VMs a new virtualization technique is developed. This technique is called *Light Virtualization* and remove the role of the hypervisor, since all the VMs (now called containers) are now directly managed by a daemon running directly on the host OS. This, in combination with a layered fs, enables to avoid adding up too much overhead in the management process of the containers.

All the thesis work is done under the European funded project called SECURED [3], that has the main aim to bring security to all device regardless its processing power. In a classical infrastructure all the security software runs directly on the end devices, like PCs, SmartPhone, Servers, etc. The problem of that infrastructure is that most of the time devices with low computation power, like the IoT ones, will not run any security application and in consequence they are highly vulnerable to cyber attacks. In order to resolve this problem, the SECURED project, proposed a new architecture where all the Security Applications runs inside a single node called *Network Edge Device* (NED) [4]. As the name says, it is the device that stays at the edge of the trusted network, like a router. The NED works with a technology called *Network Function Virtualization* (NFV) [4], that uses a light virtualization technique, and it needs to be trusted. The main aim of the thesis work is to introduce the Remote Attestation procedure on the figure of NED.

At the time of writing of this thesis, some commercial solutions are available but no one of them can introduce run-time Remote Attestation as needed by the SECURED project. The thesis work start up in this condition and take up the research work done by the TORSEC [5] group.

The remaining part of the thesis is structured as follow:

- **Chapter 2**: In this chapter we are going to introduce all the concept and terms useful in order to better understand the thesis work presented in the next chapters. In particular we will start by defining the *Trusted Computing* and we will then focus to the *Remote Attestation* procedure. The chapter finish with a brief introduction to the docker world, that is the platform used to provide light virtualization in the development of the thesis

- **Chapter 3**: In this chapter we will discuss the state of the art regarding the Remote Attestation process for a light virtualization technique. In particular we will start by discussing on what docker already made us available in term of security, and instead, what is its possible attack surface. We will then move on by analyzing what is the SECURED project and what are its aim, finishing up by discussing what already has been done inside the project in order to resolve the problem of Remote Attestation on a light VM

- **Chapter 4**: In this chapter we will discuss what has been done during the thesis work in order to achieve the Remote Attestation process. In particular the attention is focused on the resolution of two main problems arised during the thesis work. We will start by analyzing the solution proposed in order to resolve the issue related to the new mapping mechanism introduce by docker v1.10+ and we will finishing by analyzing what has been done in order to resolve the caching mechanism of IMA

- **Chapter** 5: This chapter represent a User Manual that will introduce all the concept and step necessaries in order to make the solution proposed work. In particular it will start by patching and compiling a custom version of the linux kernel, it will continue by installing and configuring the docker daemon while patching, compiling and installing a custom version of the docker cli. Finally are reported the step necessary in order to successfully patch and install the custom version of the Open Attestation Toolkit needed by this project
- **Chapter** 6: In this chapter is available a quick programmer manual, where all the modification done to the various project are available in a source code format. All the source code is commented so that it is easier to understand for future development on this topics
- **Chapter** 7: In this chapter we will analyze some alternative solutions like Intel Cloud Integrity Technology and Core Os. After that we will make a brief comparisons between them and the proposed solution
- **Chapter** 8: In the last chapter we will sum up all the thesis work and we will make the conclusions

Chapter 2

Background

As the use of *Cloud Computing*[1] continues to increase significantly, more and more importance is assuming the needs of a node to be able to trust other nodes. Cloud nodes are connected directly each other and to the network so it is extremely important to be able to recognize if a node has been tampered in a way that the original behaviour has been changed. From this preview is clear that computer security is increasingly assuming a central role in cloud computing. In this chapter we will introduce all the topics needed in order to understand better the architecture proposed in the next chapters. In particular in section 2.1 we will introduce the concept of *Trusted Computing* and *Trusted Platform*, in section 2.2 we will introduce the concept of *Remote Attestation* and in section 2.3 we will briefly discuss the *Docker* platform and how it works.

2.1 Trusted Computing

The *Trusted Computing* is an idea introduced in the late '90s by the *Trusted Computing Group* (TCG) [7] in order to define the concept of "trustworthiness" of an IT platform. As described by *IETF* in the *RFC 4949* [6], a system is trusted if:

> *"$ trusted system 1. (I) /information system/ A system that operates as expected, according to design and policy, doing what is required – despite environmental disruption, human user and operator errors, and attacks by hostile parties – and not doing other things [NRC98]. (See: trust level, trusted process. Compare: trustworthy.)"*

Nowadays that all the nodes are connected each others inside and outside the local network, it is not only important to protect the information exchanged by using protected channels, but also to be able to verify the integrity status of each nodes. This is extremely important because, no matter how we protect the exchanged data, but if a node is compromised, it will act in a bad way, possibly compromising the data received.

All the technologies developed toward this aim are done by the *Trusted Computing Group* (TCG) and are based on a chip called *Trusted Platform Module* (TPM).

This chip is used as the root of trust for all the *Trusted Computing* features developed by TCG. It is important to note that all the technology are not developed in order to recognize a compromised machine in the hardware component, but only in the software one.

2.1.1 Trusted Systems

The most important features that a trusted system [6] must implement are:

- **Trusted Platform Module**: The *Trusted Platform Module* (TPM) is a chip that is designed in order to be able to be tamper resistant. In that chip is stored an *Endorsement Key*, that is a 2048-bit RSA key, created by the manufacturer at the chip creation time. This key pair is used in several encryption processes, and can be used during the Remote Attestation process. The private key will never leave the TPM chip (otherwise the chip has been compromised and cannot be used anymore), while the public key is used for attestation and encryption of data sent to the TPM chip
- **Memory curtain**: Memory Curtain is an extension of the Memory Protection technique available on the OS. Curtained memory are areas of memory where very sensitive data are stored, and also the OS cannot access this part of the memory. The implementation details of Curtained Memory are not defined by the TCG group but they are vendor specific
- **Secure I/O**: All the information that transit on the system bus must be ciphered
- **Sealed storage**: Sealed Storage is a technique that bind private data stored on the machine to the private key of the TPM chip. This will allow the access to the data only to the correct hardware and software configuration
- **Remote Attestation (RA)**: Remote Attestation is the process that let a third party to be able to identify if a particular machine runs software that is not authorized or supposed to run. We will expand further this concept in the following chapters

In the basic Remote Attestation process, while requesting a proof of trust to another machine, you are also identifying that machine. In most scenario this is good, such as a banking site. But there are cases where want to act anonymously but still be able to be trusted by someone. This can be done by using the *Trusted Third Party* (TTP) that is an intermediary between the machine, and plays the role of the *Certification Authority* (CA).

2.1.2 Trusted Platform Module

The *Trusted Platform Module* (TPM) [8] is a chip that is usually inserted into the motherboard of a calculator, and it is used to support the solution proposed by the

Trusted Computing. It is possible to use the TPM via software by using a set of predefined command. The basic structure of a TPM chip is the one represented in figure 2.1. As we can see, the TPM contains a series of cryptographics coprocessor plus a series of register called *Platform Configuration Register* (PCR). The basic operation that a TPM can offer are Cryptographics operation like RSA key generation, Digital signature, hashing and so on.

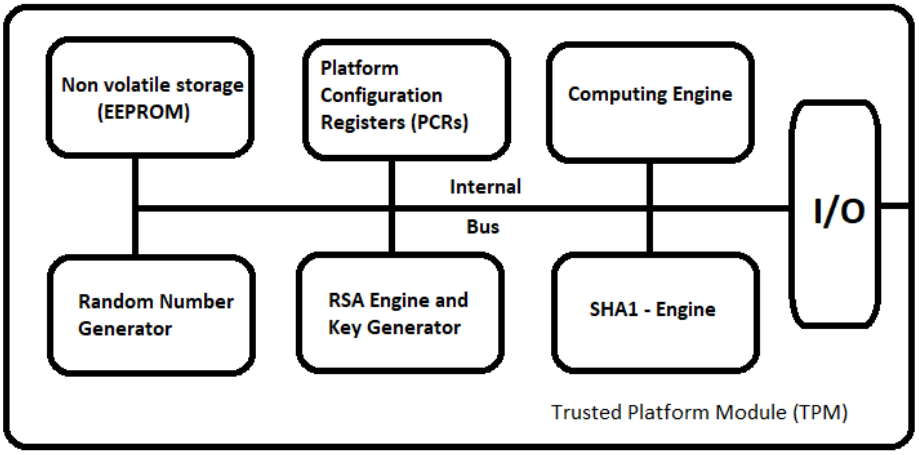

Figure 2.1. Simplified architecture of a TPM chip

As previously said, every TPM has a minimum set of 24 PCR register, 20 bytes each, that are considered a secure storage area. As described by the TCG, these registers are initialized during boot time and then cannot be erased. The only possible operation that can be done on the registers is the extension. In particular the new value of the PCR can be obtained with this formula:

```
PCR new = SHA1( PCRold || new_measure)
```

Where SHA1 is the *Secure Hash Algorithm* applied to the concatenation of the old PCR value and the new measure done. Because we are always extending the PCR, all the value of the PCR depends on the history of the previous measure. This is very important for the Remote Attestation process.

2.1.3 Chain of trust

In order to trust a virtualized node, we must trust also what is beyond the virtual node down to the hardware. For doing that we must establish a so called *Chain of trust*. This is done by validating each piece of hardware and software from the bottom layer to the one that we are interested in. The validation process is done by using a cryptographic signature. In particular each component will load only components that have a valid signature. With this procedure we can trust the last components because the previous component will not have been loaded it if

it's signature was not valid, and we can trust the previous component because, in turn, it will not be loaded if its signature check failed. However there is a special component that is called the *trust anchor* or *root of trust*. This special components is the first one that is loaded and it's certificate is a self signed certificate called *root certificate*. This certificate will not be checked by anyone so we need to trust in faith the root anchor!

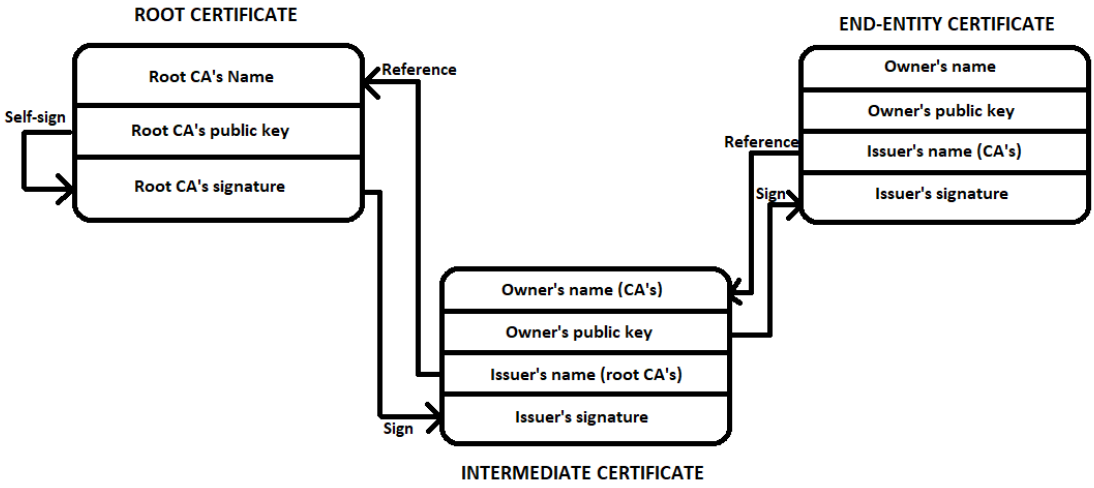

Figure 2.2. Root certificate used in the chain of trust procedure

2.1.4 Integrity Measurement Architecture

The *Integrity Measurement Architecture* (IMA) [9] is a linux kernel module that is useful for obtaining integrity measure. IMA is developed following the rules specified by the TCG (see section 2.1) and it is available into the linux kernel since version 2.6.30. The only requirements for using IMA is the availability of a TPM on the machine that is nowaday widely available in most computers.

IMA is composed of several modules that made available different functions. In particular some of them are:

- **Collect**: measure the file before it will be loaded into system memory
- **Store**: add the collected measure to a list at kernel level, and if a TPM is available in the system, extend the PCR10 register with the new measure
- **Attest**: sign the value in the PCR10 register with the TPM private key, in order to do a remote attestation procedure of the measure list
- **Appraise**: locally validate a measure with a good known value
- **Protect**: protect the security attribute in the file extension from attacks

The first three functionality are available since the first version of IMA, instead the other two where added later and are now available under the *Extended Verification Module* (EVM) [9] kernel module that is not a part of IMA itself.

During the execution IMA maintains a list of measure saved on a file inside the securityfs of the kernel. The list is available into an ASCII encoded file called *ascii_runtime_measurements* and in a binary format inside the *binary_runtime_measurements*. IMA maintains also an aggregation of all the measure done inside the PCR10 register of the TPM.

In most linux distribution IMA is disabled by default, so in order to make it works it is necessary to boot the kernel with the *ima_tcb* flag enabled.

2.2 Remote Attestation

The *Remote Attestation* (RA) is a process in which a certain node request to a remote system to report his integrity status. The purpose of this process is to be sure that the remote system is a trusted one, so it has not been tampered in order to misbehave.

When a RA request happens the requester sends a *TPM_Quote* request to the node to be attested and receives back the set of PCRs register of the TPM of the platform to be attested. This data are digitally signed with an *Attestation Identity Key* (AIK) key directly generated by the TPM and a nonce is added to them in order to avoid attack of type replay. The data sent to the attester are often encrypted so that only him can see the status of integrity of the machine.

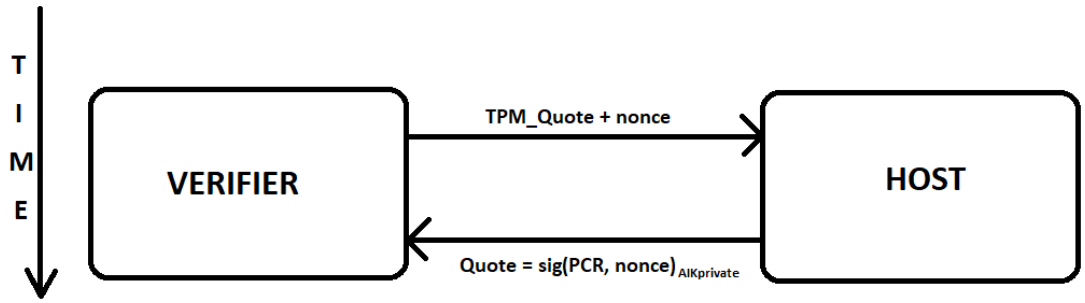

Figure 2.3. Temporal flow of Remote Attestation interaction

Once the data are sent to the attester, they are analyzed as follow:

- Verify that the AIK key used to sign the response is signed by a valid Privacy CA
- Verify the signature of the response

- Verify the measure list with the one available in the trust db and report the integrity status.

An example of Remote Attestation can be seen in figure 2.3

2.2.1 OAT Framework

Open Attestation Toolkit (OAT) [10] is a framework developed by Intel in 2010 that provides an SDK that enables us to develop a Remote Attestation service for the cloud infrastructure. In the SDK are also available a set of API that let us communicate easily with the *HostAgent* and the *Appraiser*. A simplified version of the OAT architecture is represented in figure 2.4. The only requirement of the OAT framework is the presence of a TPM on the machines to be attested.

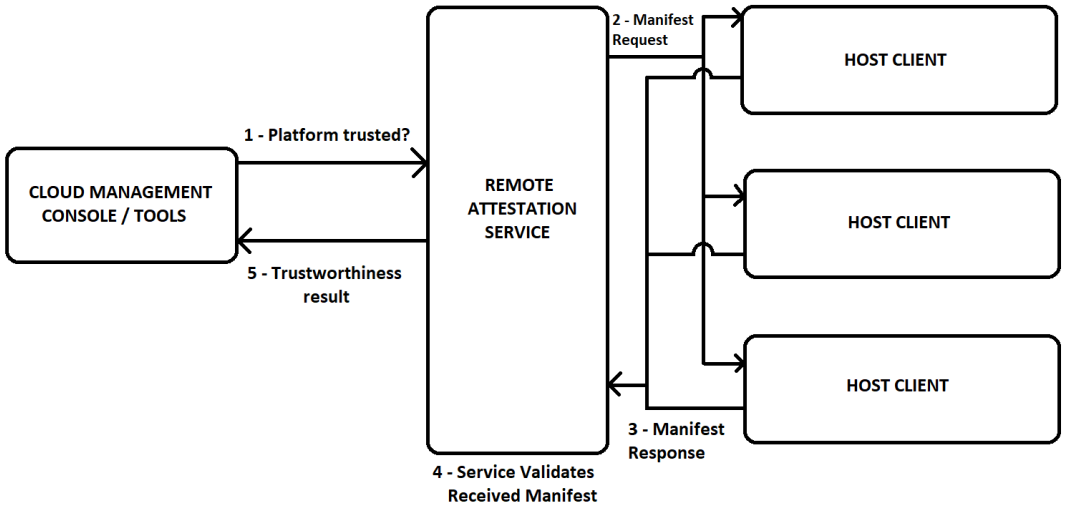

Figure 2.4. Basic interaction scheme using the OAT framework

It is compliant with the TCG methodology and implements all the functions that needs to be available for RA service like:

- **PrivacyCA**: the one that generates the certificate for the AIK key used to sign the measure associated with the platform to be attested
- **HostAgent**: a RA client that runs on the platform to be attested and that listen for RA requests
- **Appraiser**: is a sort of broker that receives and manages all the RA request of the cloud of interest
- **WhiteList DB**: it's a db in which are saved all the measure that are considered valid for a certain remote entity

When a new RA request is presented all the HostAgent creates an *Integrity Report* (IR) that is a valid XML file divided in two sections. In the first section are reported the actual values of all the PCRs registers of the TPM, instead in the second one are available all the measure done by the HostAgent. The IR is then sent back to the Appraiser that validate the status of each HostAgent by means of the *VALIDATE_PCR* and *COMPARE_REPORT* operations. With the first one the appraiser is checking the PCR values of the current IR with the value available in the last one. With the *compare_report* operation the attester is checking the validity of each measure with the ones available in the *WhiteList db*. This two operations are completely separated since version 1.7 so it is possible to avoid the *validate_pcr* and use IMA instead. With IMA it is possible to check only the PCR10 for each measure, since it contains an aggregation of all the measure done on the system.

As we seen OAT is developed in order to simplify the use of a RA process in the cloud management operations. However it is designed to work with hardware machines and not with Virtualized environment so some changes are needed to the framework in order to support the latter ones.

2.2.2 Integrity Verification Proxy

One possible solution in order to achieve RA on a virtual machine is the use of a special component inserted directly inside of the hypervisor of the vm, called *Integrity Verification Proxy* (IVP) [11]. The IVP must be a trusted component since it is the one that gave us the trusting information for each virtual machine.

With this solution the attester for the vms is directly the physical node on which they runs, so we must perform the Remote Attestation process for the physical hardware and for the proxy using the classical RA process described before.

An *Integrity Monitor* is the core of the system. It is a service with the role to directly manage all the execution phases of all the virtual machines inside the host. During the execution of the vm, the Integrity Monitor verifies the integrity status of the vm by checking that the criteria defined during the boot process are respected. During the boot of the vm some data are collected regarding the initialization parameters and the kernel image loaded. If a change in the configuration of the vm is detected the vm is stopped and the new configuration is checked. If all the new parameters meets the requirements given by the client then the vm is started again. If something wrong is detected the vm remain stopped and all the communication of the vm are interrupted in order to avoid the corruption of other nodes. When a client needs to connect to a particular VM it will first establish trust in the host node and then it will ask to the proxy service for a connection to the desired VM. This connection will be open if and only if the VM is trusted following the rules given by the client. A basic scheme for establishing a connection with IVP is represented in fig 2.5

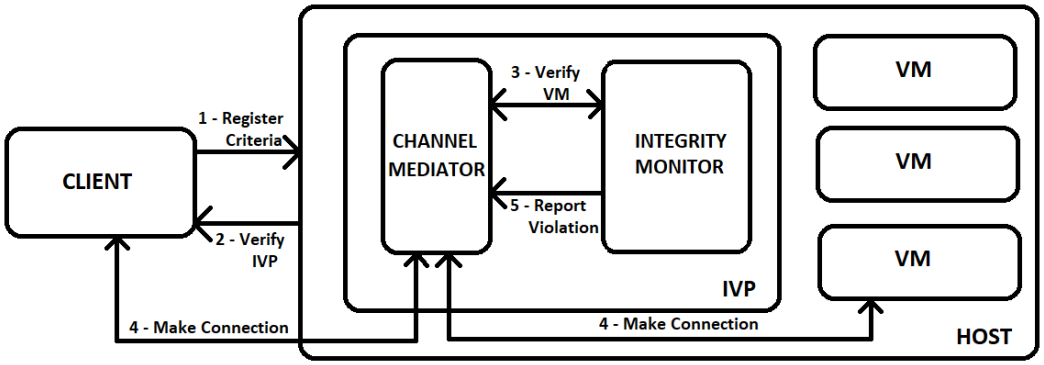

Figure 2.5. Connection mechanism using the Integration Verification Proxy architecture

The only problem with that solution is that in order to support real time integrity management the vm must be ran in debug mode and this creates a very strong bottleneck for vm performance.

2.2.3 Light Virtualization

The introduction of Virtualized Machine brings greater flexibility and scalability to cloud and now they are widely used in most of the cloud scenarios. With that type of infrastructure it is possible to introduce the RA with the solution proposed in the previous sections. However this technique introduce also a very high latency during the creation and the management of a VM. This problem can be partially solved by using a new technique called *Light Virtualization*.[2]

The difference between this two technique is that the light will no longer use a component like the hypervisor in order to manage and isolate the various vm but instead implements this functions by directly exploiting the kernel capabilities. This reduce significantly the latency of the management operations. A basic architecture can be seen in figure 2.6

As it is possible to see in the figure, the virtualization engine runs above the Host OS like a normal application and manages every VM, that in the context of Light Virtualization are called *Containers*. This is done by exploiting the library directly available on the Host Operating System. On that engine then runs all the various containers with all the application and necessary dependencies installed directly on it.

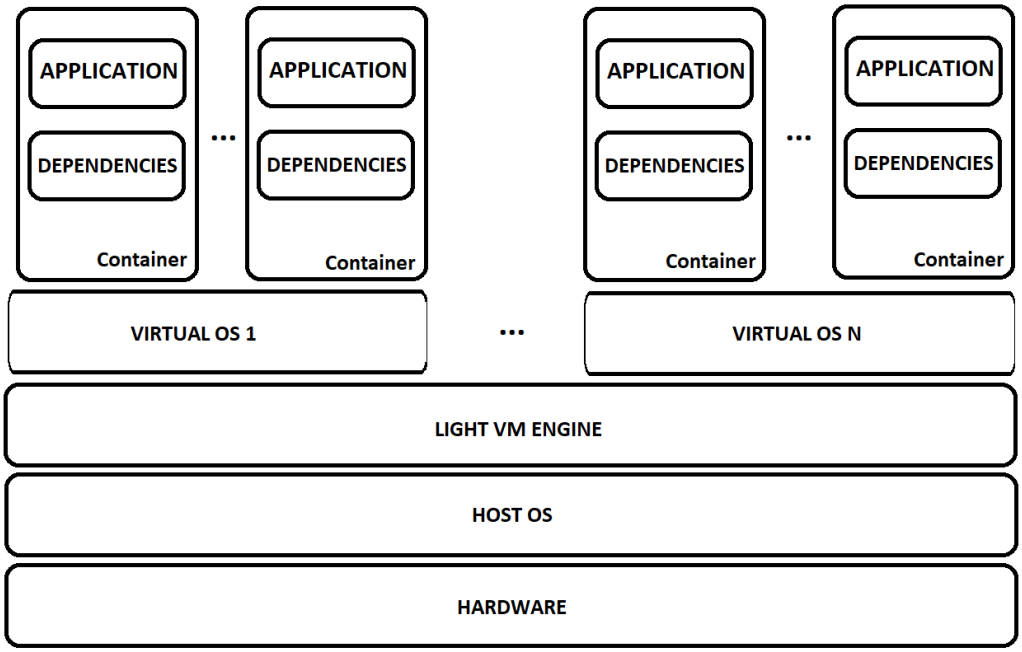

Figure 2.6. High level scheme of a Light Virtualization architecture

One downfall from switching to the light Virtualization is that, since it is a very new technique, no solutions are available for introducing the Remote Attestation process on that infrastructure. This thesis work is focused on introducing a RA process for a particular Light Virtualization engine, called *Docker*. This engine will be described briefly in the following sections.

2.3 Docker

As we have said the main aim of the thesis is to introduce the Remote Attestation process on a light virtualization scenario. For this purpose the chosen virtualization engine is *Docker* [12]. It enables the virtualization mechanism without the needs of an hypervisor. Each Virtual Machine is loaded inside of a container, that unlike a standard VM, will not contains the full Operating System but only the application and the dependencies that are needed.

Container management and isolation is done by exploiting some kernel functionality like *namespaces* and *cgroups* of the linux kernel. In particular the last one allow to allocate and manage hardware resources for each container as well as manage his status. Instead namespace are used to guarantee the isolation between the filesystem and processes of different containers.

Docker containers are very flexible since they are not hardware dependent and can be easily transferred from one node to another. Also, since they not contains the full Operating System, they do not introduce a big overhead on the hardware and are also smaller than a standard vm. It is possible to run a docker client on

almost every host Operating System, however the docker server must be installed on a Linux based system.

2.3.1 Containers

As we seen before in Docker a Virtual Machine is called *Container*. Each container can be constructed by starting from an image and inside them one or more applications with all the dependencies can be started. Docker provides to each container all the hardware resources and isolation that it needs.

Docker enable us to control each container with a set of standard instruction available in the docker *Command Line Interface* (CLI) [13]. A set of useful ones are:

- **docker attach**: attach the local I/O to the specified running container
- **docker cp**: copy files between a container and the local filesystem
- **docker create**: create a new container
- **docker exec**: run a command inside the specified container
- **docker inspect**: return low level information of one or more containers
- **docker kill**: kill one or more containers
- **docker pause**: pause all processes related to the specified container
- **docker restart**: restart the specified container
- **docker run**: run an image inside a new container
- **docker unpause**: unpause a previously paused container

For example when we run the command *docker run*, the following operations are done by the docker daemon in order to load the new container:

- search for the specified image, locally or online
- create a new docker container and load the image in it
- add to the image a new fs layer with the writing privilege
- create a new network interface and assign to it a new IP address
- load the specified startup process

Each container is loaded by the docker daemon at the same ways, and this gave Docker a great flexibility. Since all the dependencies for a particular application are preloaded inside the container, this one can be easily transferred to a different node without the need to be worried about the host environment.

2.3.2 Images and repositories

Every container is loaded starting from an image. A Docker image is a particular layered filesystem structure in which different layer, with different access privileges, are layered one on another [14]. All this level are bound together by using the Linux *Union filesystem*, as it can be seen in figure 2.7.

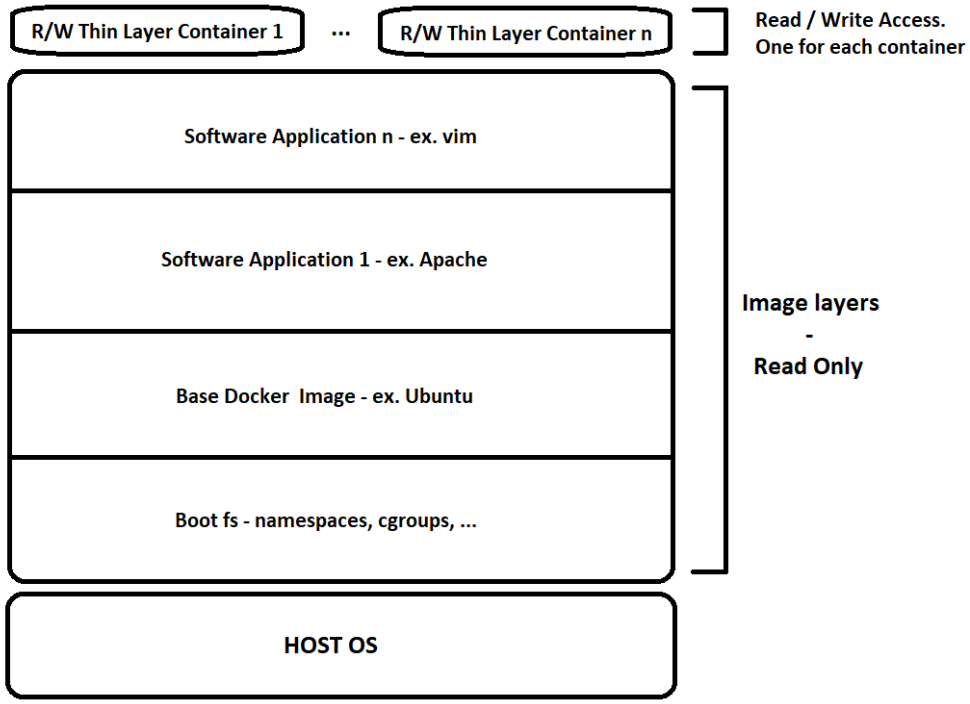

Figure 2.7. Structure of the layerfs used by docker

At the bottom of the structure is available a filesystem called *bootfs*, that is used to boot the container. On the bootfs there is another file system called the *rootfs* and it represents a particular linux OS. The rootfs is in common with more containers and it is the ground on which all the other layers are built. All this structure is not available in writing mode to the containers. Instead, for each container, a new writing layer is added on top of the framework, and it is used to maintain all the modification that the container does to the base virtualized OS. Every time a file of the virtualized OS needs to be changed by the container, the file is copied on the top layer and it is saved here. When it needs to be loaded the daemon will look first on the top layer for a modified version and then it will goes to the base one if it is not found.

It is possible to build a docker image by using a *DockerFile*. This file is a script that allow us to start from a base image and build on it our container. All the

docker images are saved inside *registers* or *repository*. Some of them are publicly available, while other are private. In particular we can identify:

- **Top-level repository**: is managed by docker himself or by development teams (like Fedora and Ubuntu) and represent the starting point for constructing the personal images. All the images posted here are secure, tested and updated frequently
- **User repository**: it is the personal repository of a certain docker user. The images posted here are public or private depending on the will of the user

2.3.3 Storage Driver support

Docker rely on the Union filesystem in order to launch and manage the containers. Every layer inside this structure is identified by means of a *Universally Unique IDentifier* (UUID). In Docker a UUID is a random hex number on 256 bit, however it uses only 12 hex char (48 bit) to identify a particular image or container. More than one container can share the same underlying image, and that allow docker to quickly create new container without the need to copy all the layer stack for each new container. However in order to keep all the modifications done by each container, a new layer is created and the writing privileges are given to the container.

All the this layers are managed by a *Storage Driver* [15] and they can be accessed through the host filesystem typically at the position */var/lib/docker*. The Storage Driver maintains all the layer structure and it execute the *copy-on-write* operation when a file need to be changed by a container. With that operation a new copy of the same file available in the base image is created in the upper level where the container can access it and modify it. The copy available in the base system is never modified and all the modification done by the container are saved and accessed only on his upper layer.

The way in which the layer structure is managed largely depend on which storage driver is in use on the docker daemon. In particular, Docker, supports a certain number of storage driver and the default one change in base of the host OS that is being used. However it is possible to configure the daemon in order to use one particular storage driver. For the purpose of this thesis we will now focus on the *Device Mapper* storage driver.

Device Mapper is the second storage driver for which Docker provides support for. Basically it maps physical blocks with virtual blocks of higher level. Every virtual device is created starting from a base pool that is created during the initialization phase. It memorizes all the images and containers to virtual devices by using the copy-on-write technique, so each file, even if available in the host system, can be seen as it belongs to a different virtual device. Device Mapper works at block level and doesn't manage each single file, so when a file is required it will copy the entire block the file belongs to.

Chapter 3

Remote Attestation on Light VM

This chapter will present the security features available in docker and the state of the art for the *Remote Attestation* process on *Light Virtualized Machine*. In particular in section 3.1 we will explore the security feature available in docker, ranging from the *kernel namespaces* and *cgroups* to the *Docker Content Trust* and finishing up analyzing its attack surface. Instead in section 3.2 it will be presented the SECURED project and its aim, while in section 3.3 it will be presented the modified version of the *OAT framework* for docker done by the TORSEC group.

3.1 Security in Docker

When using a software solution, one of the principal aspect that needs to be considered is the security one. This is especially true when considering a *Light Virtualization* environment like *Docker*. In particular we must be aware of what are the security feature already available in docker and which are the possible security fault that needs to be considered and, possibly, corrected [16]. In the case of docker the analysis can be divided into two macro areas:

- The attack surface provided by the docker daemon itself
- The attack surface and the security features available in docker and of the technology on which is based, like the namespaces and cgroups

In the following sections we are going to discuss in detail this two macro areas in order to assess and discuss the possible vulnerability and solutions.

3.1.1 Kernel namespaces and control groups

In order to work, docker exploit some linux kernel capabilities like the namespaces and the control groups. Starting with the *Namespaces*, they are a feature of the Linux Kernel that let the user to virtualize system resources to a group of processes. In the latest kernel version (4.10 at the time of writing) it is possible to have seven different kind of namespaces and they all works in the same way by assigning the

resources that they are carrying only to the group of processes they are targeting. The target processes are able to see only the resources from the namespaces they are assigned to.

Docker uses this functionality in order to create isolation between each containers. In particular, for each container, Docker create a set of different Namespaces. The most important one for security purposes are the *Process ID* (PID) namespaces, which have the ability to provides to a container an independent set of process IDs in order to provide isolation at the process level. For providing isolation to the network level, Docker creates also a namespace of type network. This type of namespace virtualize all the network stack, and provide a private set of IP, routing table, socket listing, firewall and other network related resources for each group of processes. When a new network namespace is created it is available only a loopback interface. For that reason docker will automatically creates a virtual network device for each container. However if the containers are configured properly they can still be able to reach each other via network interfaces.

Control Groups (cGroups) is a functionality available in the linux kernel since version 2.6.24 that enables the user to limit the resources available to a group of processes. Docker uses this functionality in order to equally distribute the available resources to the various containers in a way that no one can take all the hardware resources available. With this technique it is also possible to avoid attack of type *Denial of Service* (DoS) since a particular container can at most take control of a fraction of the hardware resources available in the host system.

To sum up, Docker uses namespaces and control groups kernel functionality in order to provide containers isolation and to assign the physical resources to the various containers. This solution provides a good level of security, but it cannot reach the security level provided by an hypervisor in the classical virtualization realm. For this reason it is necessary to rely on other features in order to provide a stronger security layer.

3.1.2 Docker Kernel capabilities

Each container started from Docker has not access to all the feature available. This means that they don't have support for a full root access. It is possible to do that since most of the critical feature (that normally requires a root access) are managed directly by the daemon and therefore externally to the container. For this reason the root inside a container is just a fraction of the real root on the host system. Docker kernel capabilities works as a whitelist, so only the permission specified are granted to the root user inside the kernel. This feature is very important for a security standpoint, since even if someone manage to get a root access inside a container, it is a very restricted operations that it can do, and it is very unlike that it manages to harm the container or escalates to the host system.

Aside from the security features directly implement in docker, it is possible to strengthen the docker installation by using other system-wide linux kernel security feature. For example it is possible to access control policies by using one of the many access control template available in linux or it is possible to use some docker

template for *SELinux* or *AppArmor*. From version 1.10 of docker user namespace are directly supported in the daemon. This means that is possible to maps the root user of a container to a non root user outside the container helping to mitigate the risk of a breakout.

3.1.3 Docker Content Trust

We have seen in section 2.3.2 that it is possible to build a custom docker image by downloading some base images using the docker registry. But how we can be sure that this images are not corrupted? For solving this problem a new feature called *Docker Content Trust* [17] was developed. This feature allows the final user to verify both the image integrity and the publisher of the images downloaded from a register. It works by enforcing a series of digital signatures to each image uploaded to the repository. All the integrity checks are then done at client side.

However it is possible for a specified publisher to have both signed and unsigned images. In particular this choice can be done when a new image is pushed to the registry. By default Docker Content Trust is disabled on client side and can be enabled by settings the environmental variable *DOCKER_CONTENT_TRUST=1*. When this feature is enabled, the client is able to see, and therefore run, only the signed images, making the unsigned ones invisible to it. Instead if this feature is off the client is able to see and run both the signed and unsigned images. Docker Content Trust is useful also to prevent attack of type replay, where an old version of the same image is presented to the user in order to exploit some known vulnerabilities available in it.

All this functionality are granted by using different keys. Some of them are stored on the client, while other are stored directly online. All the keys are generated by the *root key* that is created the first time a developer does a push of a signed image to the repository. This are the available keys for the Docker Content Trust technology:

- **root key**: it is the root key that is used to generate all the other key necessaries to use the Docker Content Trust mechanism. It is created once when the publisher upload its first signed image. It is saved offline in the client of the publisher and must be backupped frequently since if it is lost it will be very hard to recover

- **targets**: it is used in order to sign image tags and to manage delegation. For this reason is called also the repository key. It is stored in the client of the publisher

- **snapshot / tagging key**: it is used to sign the current collection of image tags, in order to prevent mix and match attacks. It is stored on the docker authentication server

- **timestamp**: it is used to sign the timestamp on which a particular image is created in order to guarantee freshness of the image to the client. It is stored on the docker authentication server

- **delegation**: it is used to allow other publisher to sign your image without the need to share the root key. Prior to version 1.11 they were stored in client side. After that version they are stored directly on the docker authentication server

In fig 3.1 it is available a graphical representation of all the keys available in order to manage the *Docker Content Trust* mechanism.

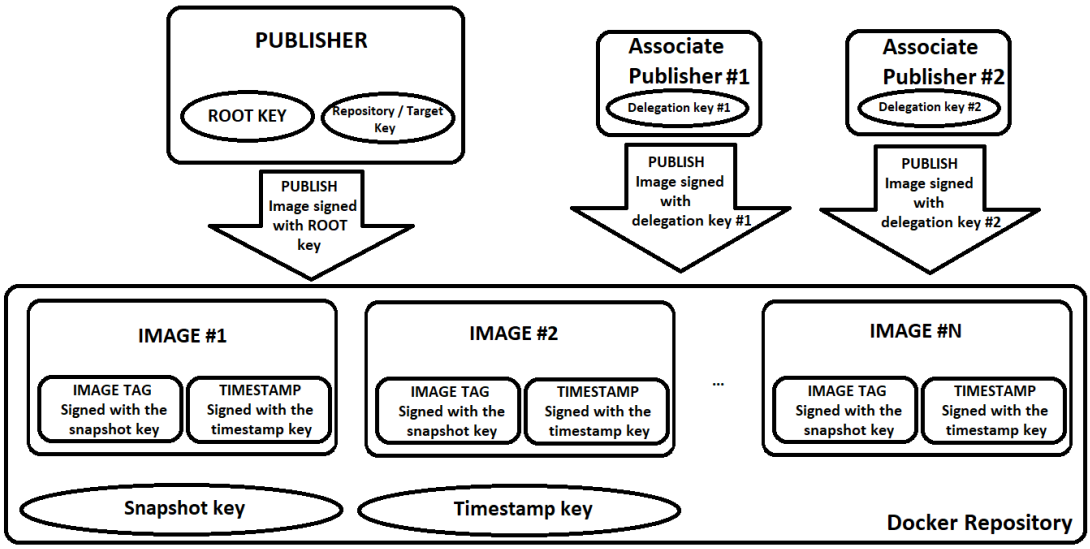

Figure 3.1. The various keys available in Docker Content Trust and how they are managed

3.1.4 Attack surface

As every piece of software available, also docker is not immune to security risk [16]. One of the main risks of docker is that the daemon needs the root privileges to run. This requirement comes from the needs of the daemon to runs some high privilege operations on the host system in order to provide some powerful features. In particular it is possible to share a folder between the host system and one or more containers. The problems is that, the container, by default has no limitation in the access of that folder tree, this means that a container can modify this portion of the host filesystem without restriction. Also no one prevents to share the root position of the host filesystem with the containers, making this a very bad choice for security purpose.

Another potential threat for security comes from the fact that docker daemon exposes a set of API on a socket. Before version 0.5.2 the API were exposed to a TCP socket binded on 127.0.0.1, but since this solution was prone to request forgery attacks, the API are now exposed on a Unix socket so that it is possible to use the standard UNIX permission checks to limit and control the access to the socket.

It is possible also to expose a set of REST API via HTTP. However it is highly recommend to secure them by using HTTPS or some protected tunnelling method such as a VPN.

We have seen that Docker provides an integrity service while using its repository by means of the Docker Content Trust technology. However it is possible to load some images directly from the disk of the machine by using the command *docker load*. This exposes the docker daemon to some input vulnerabilities and for such reason from version 1.3.2 all docker images are extracted in chrooted linux subprocess. This is a first step towards privilege access separation but some more work in that direction is needed.

As seen docker implements a lot of security features but lacks an integrity check at runtime. For this reason in the next chapters we are going to analyze more in deep the work done in order to implement this feature.

3.2 SECURED project

All the thesis work has been developed under the SECURED [3][4] project. It is a three years project funded by the european union and it is entirely related to cybersecurity. The thesis work is just a small part of the entire project, so in the following subchapter we will introduce the aim and the general infrastructure of the project.

3.2.1 Introduction

In the last decade we have started to access the network using different kind of devices that range from PCs to Smartphone, from smartTV to the general IoT. Since, until now, the security is done on the end device, this introduce a security problem. Every device has different hardware, computation power and software and since a full security suite requires some computational power, not all devices are well protected from possible attacks. Moreover not always the devices are connected to a secure network, i.e. not always are presented security related network hardware such as a firewall or an IDS. For example when accessing the network from a cafè network we are not protected from attacks.

In order to resolve this issues, the main aim of the SECURED project is to move the security equipment form the end device to a trusted node in the network so that all the end user device and application are completely unaware of the security issues but they are equivalently safe. This is very important in applications such as IoT where, due to the low power requirements, there is not enough power to secure the device, so each device is vulnerable.

3.2.2 NED and NFV

The main idea on which is based the research work is to move all the security components from the end user device to a trusted node inside the network. This

node is called *Network Edge Device* (NED) [4] and as the name says it is located at the edge of the network (it can be easily the home gateway, a WiFi access point or some other network equipment). Since it will run all the security applications and protocols, it is very important to verify the integrity status of this node. This can be done with the standard Remote Attestation technique introduced in chapter 2.

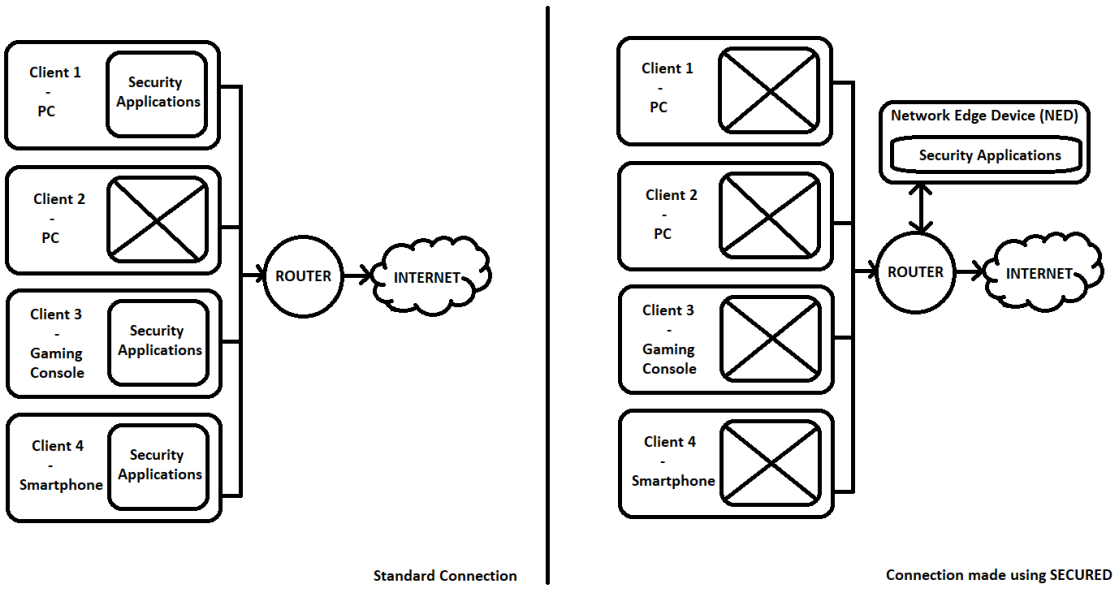

Figure 3.2. Standard connection vs connection made using SECURED

It is important to say that it is possible for the NED to have different security profile for different users. NED, for each authenticated user, maintain a security configuration that is used to run a series of *Personal Security Applications* (PSA) inside the virtualization ambient related to that user. It is possible to ingoblate the NED directly inside the *Network Edge Device*, for example a router, or to separate the *Network Device* and the *Security Device* into two different devices connected with a secure channel.

As said, when using the NED, there is the possibility to run all the *Personal Security Application* inside a virtual machine, that in the case of a *Network Device* it is based on the *Network Function Virtualization* (NFV) technology. Since the network device doesn't have a lot of power, it is possible to run all the *Personal Security Application* using a Light Virtualization technique, such as docker, instead of a more heavier standard Virtual Machine. Since the NED is a critical node in that infrastructure it must be trusted, so it is important to be able to remotely attest it.

The benefit introduced by using a Network Function Virtualization technology are the following:

- **No more dedicated hardware**: by using NFV we doesn't have the need to buy different hardware for each Network Device that we need. Instead it is sufficient to have a general purpose hardware with a good network interface and install all the network function that we need on it

- **Greater scalability**: if in a certain moment we need to have a particular network device, it is sufficient to install it directly on the existing general purpose one

- **Personal services**: since all the network devices are virtualized it is possible to have several instances of them in order to personalize the level of services and protection offered to each connected user

3.2.3 Architecture

The security architecture developed inside the SECURED project is constructed around the presence of the NED. In particular all the security applications will now not run on the end user devices but on a trusted node at the edge of the network. Also there is no needs to have multiple physical hardware devices in order to cover different network and security functions since they are all virtualized on a generic hardware by using the NFV technique. In order to create a connection, each end user device must follow this step:

- **Trust the NED**: the client needs to put trust in the NED figure, by using a Remote Attestation procedure described in chapter 2 or, in case of Light Virtualization machines, using the architecture developed with this thesis work

- **Client authentication**: once the client has put trust into the NED, it's time for it to authenticate itself

- **Personal Security Application loading**: the NED goes into its Application and Policy Repository to fetch all the configuration for that particular client

- **Personal Execution Environment**: the NED, with the configuration fetched in the previous step, creates a *Personal Execution Environment* (PEE) by using a NFV technique. From that moment onward, the client can securely connect to the rest of the network by passing to its PEE

A graphical scheme of a connection made using the SECURED project is available in fig 3.3

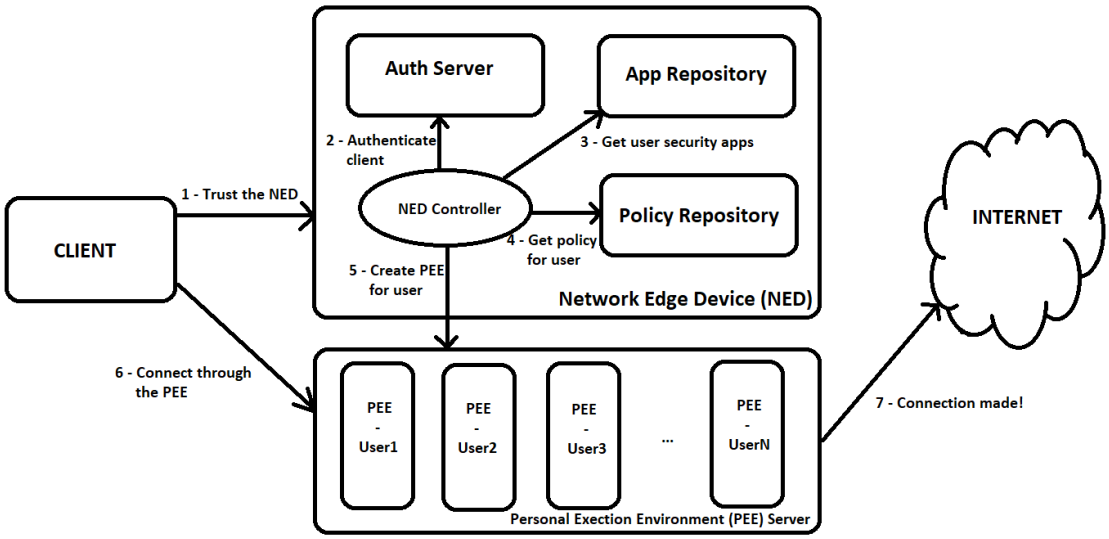

Figure 3.3. Connection procedure using the SECURED project

3.3 OAT Framework for Docker

We have seen that in order to made available the *Remote Attestation* process also for a light virtualization technique some new features must be developed. Some steps in that direction has already been taken by the TORSEC [5] group at Polito. All the project is based on the *Open Attestation Toolkit* framework [10], that as we have seen in chapter 2 is a framework developed by Intel in order to make easier to implement the Remote Attestation process. However this framework, out of the box, will work only with physical nodes so some modification must be put in place in order to adapt it also to a light virtualization engine like docker.

The architecture chosen in order to make it works with docker is the one represented in fig 3.4. Basically the OAT framework is composed by three different components that works together:

- **HostAgent**: A client application that runs on the node that needs to be attested. It is interfaced with the node TPM by using the TCSD daemon. When a new Attestation Request arrives, it produces an Integrity Report (IR) that contains all the value of the PCRs of the TPM of the machine and a list of all the measure done on the system
- **Appraiser**: It can be seen as the "server" of the application. Is the one that has the role to actually verify the integrity status of each node by analyzing in detail the IR produced by each one
- **WhiteList database**: It is a database in which are saved all the valid measure of each node. It can run on the same machine of the Appraiser or in a different one

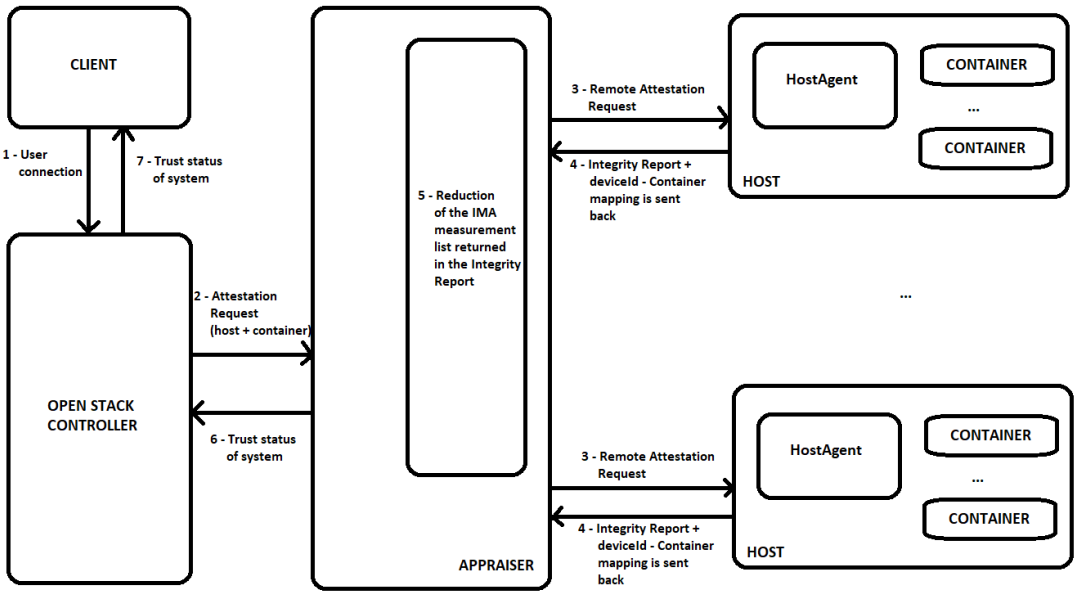

Figure 3.4. OAT architecture for docker

In order to make the system work with docker some changes were needed. In particular all the system must be updated in order to make it works for different container on the same host. This has been done by modifying the Host Agent in order to identify for each measure the container of membership and extending the Integrity Report in order to contain also a list of active container. The changes done to the Integrity Report must be reflected also to the Appraiser in order to correctly analyze it. It is important to note the in order to verify the integrity status of the report the project will not use the standard OAT procedure described in chapter 2 but it will use the extension method of the TPM.

The project described in this section is surely very interesting but it runs only for docker version less then 1.10 (since from that version docker changed the way containers are mapped to the virtual devices), and it has some security flaws that needs to be solved. So the thesis work is based on the research work done by the TORSEC group and its main aim is to make it works for docker version 1.10+ and to identify and resolve security and functional issues. In the remaining section of this chapter we are going to analyze in detail the major changes made to the HostAgent and the Appraiser by the Polito research group before the thesis start.

3.3.1 HostAgent

The *HostAgent* is a client application that runs on every node that needs to be attested. Once in execution it will remain listening for Attestation Request by the appraiser. All the communication between this two components is done by a web services exposed by the components. When a new Attestation Request comes it will

produce an Integrity Report that contains all the data needed in order to correctly test the integrity status of the node.

An *Integrity Report* is a file in XML format that, as defined by the TCG, contains the following fields:

- **Quote**: contains all the data returned by the Quote procedure done on the host TPM. In particular it contains all the values of the various PCRs plus a bunch of information useful to verify the TPM signature done on all the data returned by the IR
- **PcrValue**: contains the value of the PCR specified by the PcrNumber attribute at the moment of the IR request
- **TpmSignature**: it is the signature of the TPM done on the returned data in the IR, plus a series of information useful to identify the type of signature done by the TPM
- **SnapshotCollection**: is a list containing all the measure done by IMA into the host that needs to be attested
- **SimpleType**: it is the single measure done by IMA. In particular it contains the name of the template used, the digest value calculated by IMA and a base64 encoded string with all the other fields of the measure
- **ComponentID**: It is an element that is inserted inside the SnapshotCollection, and it contains a series of information useful to analyze the measure list reported after it

In order to make the project usable also with docker, some information has been added to the *ComponentId* field of the IR, and they are:

- **Container**: it is a mapping between the virtual device id and the Docker container id that is used by the appraiser to identify for each measure which is the corresponding container
- **Host**: it contains some information of the host system to be attested. In particular it makes available a list of all the device id associated to the host system

However, in order to correctly compile the IR, some changes are done also to the IMA module of the linux kernel. This one has the role to measure at runtime all the file that have been loaded into memory. In particular a new template has been defined in order to include also the deviceId of the file measured.

3.3.2 Appraiser

All the changes made to the HostAgent must be reflected also to the Appraiser. The role of the Appraiser is to verify the integrity status of all the nodes registered to it.

This is done by executing a Remote Attestation request to the node to be attested and analyze the Integrity Report that is sent back.

The parser has been changed in order to consider the new data inserted into the Integrity Report and a new type of analysis, called *cont-check*, has been added to the project. This new analysis work exactly like the standard *load-time* analysis except that it will analyze only a subset of measure that are the one related to the container ids passed as parameters. However the IR will contains all the measure done on the host since the appraiser needs to replicate all the extension operations done to the TPM in order to verify the integrity of the report. If the list of measure were shrinked directly by the host agent, the integrity check on the report will fail.

Chapter 4

Project changes

In this chapter will be presented the thesis work done in order to resolve the problems described in the previous chapters. In section 4.1 the work is presented in a general way. In section 4.2 it will be presented the current version of OAT for Docker with its problems and how it will be changed in order to solve them. In section 4.3 it will be presented in detail Docker and the new command developed for the use inside the architecture. In section 4.4 it will be described the IMA module and how it works, and the solution proposed to solve the caching mechanism.

4.1 Introduction

The thesis work is developed under the SECURED project. The work is concentrated on the figure of the NED. The main aim of thesis is to update the project in order to makes work with the newer version of docker and to resolve some critical problems that were encountered during the previous developments. In particular the work start off by analyzing what is already available and what are the changes in the latest version of docker. As described in chapter 3 one key point is to use device mapper as a storage driver. This will let us links every measure done by IMA with a specific container, by making a mapping between the *containerId* and the virtual *deviceId* created by *DeviceMapper*. From version 1.10 of docker, some changes are made to DeviceMapper that broke this important mappings. In order to be able to use this project with newer version of docker, a solution to this problem must be found. The solution is described in sections 4.2 and 4.3.

This project uses IMA in order to produce a list of measure for every file loaded inside the memory. By default, IMA measure the file at most the first time the file is loaded into memory, based on his policies. The problem is that the same file can be loaded by multiple containers, but it is not necessarily true that a file is allowed to run in multiple containers. With the standard behaviour of IMA we cannot identify correctly which file has been loaded into which containers. This problem cannot be solved by using the standard behaviour of IMA. The solution of this problem is presented in section 4.4.

4.2 OAT for Docker

From version 1.10, docker has changed the way it maps containers to virtual devices. In particular, before that version, it maps each container layer to a folder named after the container id. This lets the previous implementation of the project to retrieves the mapping containerId - deviceId by simply listing the folder under the */dev/mapper* path. Each folder listed corresponds to an active directory. After that version, this is not true anymore so another solution must be found.

4.2.1 Docker's DeviceMapper indirection

Before version 1.10, docker with deviceMapper, maps each container into a folder under */dev/mapper* called with the name *docker-containerId*. In thesis case the mapping containerId - deviceId can be done easily with the following method:

- Retrieve the virtual device *major number* by looking for an entry of type *dm-0* inside the folder */dev*
- List all the folder inside */dev/mapper* and for each entry that contains the word *docker* retrieve the mapping devId - containerId by parsing the name of the folder

The folder created under the path */dev/mapper* follows the naming convention *Docker-MAJ:MIN-INO-containerId* [15], where:

- **MAJ**: is the major number of the physical device on which the Docker files are stored
- **MIN**: is the minor number of the physical device on which the Docker files are stored
- **INO**: is the inode number of the path */var/lib/docker/devicemapper* on which the storage driver stores the layer of the image and containers used directly by docker
- **containerId**: is an hash randomly defined by the docker daemon that corresponds to the container Id

Unfortunately, after version 1.10, this is no more possible since docker will not map each container with a single folder inside */dev/mapper* but will maps each container to multiple folders, with an hash in the *INO* field.

It is possible to list all active container by using the command *docker ps*. For each container we can now retrieve some low level information by using the command *docker inspect containerId*.

After the changes the output of the *lsblk* command changes as follow:

```
''GraphDriver'':{
     ''Name'': ''devicemapper'',
     ''Data'':   {
                 ''DeviceId'': ''27'',
                 ''DeviceName'':
   ''docker-8:6-4853215-b665ee05b818394a5e98a294362cb955dc
                                 1a5211a412222ea840a627d72194bb'',
                 ''DeviceSize'': ''10737418240''
     }
},
```

Figure 4.1. Extract of the output of *docker inspect containerId* command

On this output we are particularly interested on the section *GraphDriver*. The field *DeviceName* will tell us the device name associated to that particular container.

Keeping that in mind it is possible to execute the command *lsblk*:

```
loop0                          7:0          0          100G       0
    loop
|_docker-8:6-4853215-pool  252:0          0          100G       0
    dm
    |_docker-8:6-4853215-b665ee05b818394a5e98a294362cb955dc
                                 1a5211a412222ea840a627d72194bb
                             252:1          0           10G       0
    dm
   /var/lib/docker/devicemapper
```

Figure 4.2. Extract of the output of *lsblk* command

In that particular case we can see that under the pool 4853215 is available a device that has a name equal to the one that we have found executing the command *docker inspect containerId*. This device has a major number of 252 and a minor of 1 so the resulting deviceId is 252:1. With that information is then possible to create the mapping containerId - deviceId by executing the command *lsblk* and using the device name as an intermediate key.

In the figure below it is possible to have a graphical representation of the two mapping procedures, before and after version 1.10 of docker.

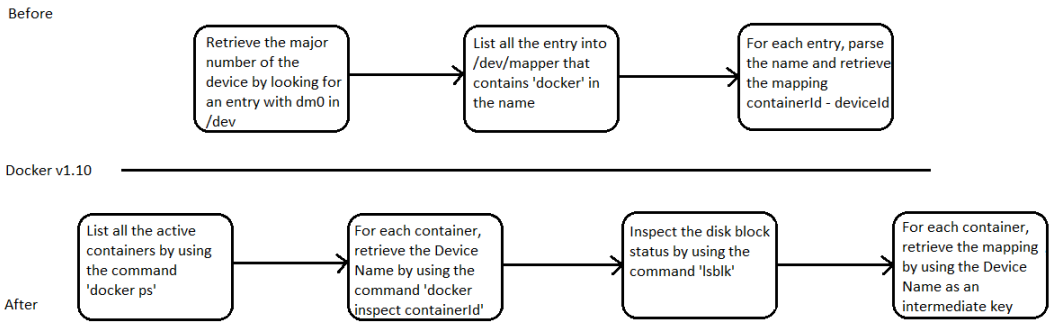

Figure 4.3. Mapping procedure before and after docker v1.10

Since the new procedure involves a lot of I/O parsing, the performances of the mapping are a bit deteriorated, so some more research on this fields are needed.

4.3 Docker

As we have seen, the proposed solution in section 4.2 increase significantly the time needed to retrieve the mapping. Since the mapping is done every time an IR is produced, further investigation must be done. By analyzing a little bit the solution proposed by putting some timers inside the code, it is possible to see that most of the time is spent parsing the Output of the various commands used. So reducing the I/O operations can lower significantly the mapping time. However this is not possible without developing a docker CLI command ad-hoc for the project.

4.3.1 A new command for docker

In order to reduce at minimum the mapping time, we need to reduce as much as possible the number of I/O operations. In the proposed solution we use three commands that makes I/O: *docker ps*, *docker inspect containerId* and *lsblk*. This three commands can be put together into a new docker CLI command. A new command is then defined: *raInfo*.

This command will follow the procedures described in section 4.2 and prints out on the screen directly the mapping *containerId* - *deviceId* for each active container. The advantage of this method is that we are eliminating all the unnecessary I/O done by the two docker CLI commands, and then we are printing only the value that we are interested in. In table 4.1 are reported the time, in ms, obtained by the two different implementations.

It is possible to see that the new solution reduces significantly the mapping time, specially when the number of active containers grows.

Number of Containers	1st Implementation time (ms)	2nd Implementation time (ms)
0	13	11
1	23	13
2	32	16
3	42	18
4	53	19
5	61	19
10	115	22
20	211	31
30	314	39
40	423	48
50	510	58
100	1020	108
150	1663	180
200	2192	265
250	2746	364
300	3297	472
350	3761	628
400	4410	798
450	4927	928
500	5506	1087

Table 4.1. Mapping times (ms)

4.4 Linux Kernel

As seen in chapter 2, the project uses the measure done by the IMA module of the kernel in order to create the IR. One limitation of the IMA module is to measure a certain file only the first time it is loaded into memory. This is done for performance reasons, but in our case this is something that needs to be changed. For example think of the following example: we have a file named *run.sh* and two containers named respectively *container1* and *container2*. *run.sh* is allowed to be loaded in *container1* but it is forbidden in *container2*. *run.sh* is loaded into *container1*, IMA will measure this action, so a row in the IR is produced. After a while *run.sh* is loaded also into *container2*: this time IMA will not measure the file again, so no row relative to *container2* is produced in the IR. In this case the file will be loaded also inside *container2* but the system will not be able to recognize this situation. Unfortunately there is no way to change this behaviour by just working with IMA configurations, but the module must be patched.

4.4.1 IMA caching policies

As we have seen IMA will measure every file that respect the IMA policy only the first time it is loaded into memory. This is done for performance reasons but this is not good for our project.

By default IMA associate to every file a structure called *integrity_iint_cache*. This structure is defined as follow:

```
/* integrity data associated with an inode */
struct integrity_iint_cache {
        struct rb_node rb_node;
        struct inode *inode;
        u64 version;
        unsigned long flags;
        unsigned long measured_pcrs;
        enum integrity_status ima_file_status:4;
        enum integrity_status ima_mmap_status:4;
        enum integrity_status ima_bprm_status:4;
        enum integrity_status ima_read_status:4;
        enum integrity_status evm_status:4;
        struct ima_digest_data *ima_hash;
};
```

Figure 4.4. definition of structure `integrity_iint_cache`

For our project, the important fields of the structure are:

- **struct inode *inode**: back pointer to the inode of the file
- **u64 version**: a field that track changes to the file if the filesystem is mounted with the flag *-i_version*
- **unsigned long flags**: the flags associated to the current file
- **unsigned long measured_pcrs**: a flag that indicates if the file has already been measured by IMA

For performance reasons, IMA also saves into an hashtable the measure already done. An entry of the hashtable looks like:

```
struct ima_template_entry {
        int pcr;
        u8 digest[TPM_DIGEST_SIZE];
        struct ima_template_desc *template_desc;
        u32 template_data_len;
        struct ima_field_data template_data[0];
};
```

Figure 4.5. definition of structure `ima_template_entry`

where:

- **int pcr**: is the number of the TPM pcr on which the measure has been extended
- **u8 digest[TPM_DIGEST_SIZE]**: is the sha1 or md5 hash of the measurement
- **struct ima_template_desc *template_desc**, **u32 template_data_len**, **struct ima_field_data template_data[0]**: are fields that indicates the caratteristic of the template used by `IMA`

In order to have a row in the measure list, a file must respect the following rules:

- Must be conform to the IMA policy defined by the user
- The *integrity_inode* associated to the file must have the flag *IMA_MEASURED* setted to 0. This is possible only if the file has never been measured by IMA or the file has changed and the filesystem has been mounted with flag *-i_version*
- The hash of the measure must not be already inserted into the hash table *iint_cache*

In order to force IMA to measure a file every time it is loaded into memory the last two points must be modified. Starting from the *integrity_inode* it is possible to force the flag *IMA_MEASURED* to 0 every time a successful measure is done. This will make IMA to think that the file has never been measured so a new measure will be produced. This is not enough, because the measure will not be printed into the list, since it was already inserted into the *iicache* hashtable. In order to force the output of the measure it is possible to skip the check on the hashtable.

In order to maintain also the standard behaviour, two new kernel boot flags, called *ima_cache1* and *ima_cache2* were added. If this new flags are set to *false* then the new behaviour will be used, otherwise it will be used the standard one.

Chapter 5

User manual

In order to be able to use the solution proposed in chapter 4, some configurations are needed. The aim of this chapter is to explain step by step what is needed in order to be able to use this solution. In particular, in section 5.2 it will be explained how to patch, compile and configure the linux kernel, in section 5.3 how to patch, compile and configure Docker and the Docker CLI, in section 5.4 how to configure the OAT Appraiser, and in section 5.5 how to configure the OAT Host Agent.

5.1 Prerequisites

The target infrastructure used while developing and testing this project is:

- **OAT Appraiser**: CentOs 7 with kernel v3.10.0-514.el7.x86-64
- **OAT Host Agent**: Ubuntu 16.04 LTS with custom kernel v4.13 - rc2

For correctly installing and configuring the project some software are required. In order to get them use the commands provided below:

- On Appraiser:

```
yum install epel-release
yum install ant trousers trousers-devel php-soap mariadb
    maradb-server python-networkx python-suds
    python-matplotlib graphviz-devel patch java-1.7.0-openjk
    java-1.7.0-openjdk-devel zip unzip gcc gcc-c++ rpm-build
    python-pip git httpd php php-mysql rpm-devel
    mysql-connector-python rabbitmq-server mod_ssl
pip install pycasa pygraphviz tornado celery urllib3 requests
```

- On Host Agent:

```
sudo apt-get install git build-dep linux-image-$(uname -r)
    fakeroot unzip
```

5.2 Linux Kernel configuration

In some scenario it is necessary to recompile the kernel in order to be able to activate IMA. Also, in the proposed solution, it is mandatory to recompile the kernel in order to be able to use the modified version of the IMA module. In this section we are going to patch and compile a custom version of the linux kernel. The following steps are done using Ubuntu 16.04 LTS as a base distribution, and the compilation result is a set of *.deb* packages that can be easily installed on any Debian based distribution. However with a similar procedure it is possible to recompile the kernel also for other distros.

5.2.1 Patching the kernel

First of all get the latest version of the source code from the official git repo:

```
mkdir custom_kernel
cd custom_kernel
git clone https://github.com/torvalds/linux.git .
```

Once the download is finished, it is time to apply the provided patch:

```
git am --signoff < kernel.patch
```

Now the kernel source code is successfully patched and it is ready to be compiled.

5.2.2 Compiling the kernel

Compiling the kernel for the first time requires some time, depending on the processing power of the machine used. So let's start! [18] First of all be sure to be in the root of the source code of the kernel by typing:

```
cd custom_kernel
```

Copy the current kernel configuration and load it:

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

Now it's time to start the compilation process:

```
sudo make-kpkg clean
sudo fakeroot make-kpkg --initrd --append-to-version=-ima-custom
   kernel_image kernel_headers
```

Once the compilation process has finished, it is possible to install the new kernel:

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

As soon as the installation is completed, is time to add the required boot parameters for the kernel. In case of grub2 bootloader:

```
sudo gedit /boot/grub/grub.cfg
```

search for the new entry, in this case the one that end with *ima-custom*, and add the following parameter:

```
ima_tcb ima_template=ima-cont-id ima_cache1=false ima_cache2=false
```

so it will looks like:

```
menuentry 'Ubuntu, with Linux 4.13.0-rc2-ima-custom' -class
   ubuntu -class gnu-linux -class gnu -class os
   $menuentry_id_option 'gnulinux-4.13.0-rc2-ima-custom+-
   advanced-35a31811-d3e1-4998-a239-e62929a8cca5' {
    recordfail
    load_video
    gfxmode $linux_gfx_mode
    insmod gzio
    if [ x$grub_platform = xxen ]; then insmod xzio; insmod
   lzopio; fi
    insmod part_gpt
    insmod ext2
    set root='hd0,gpt6'
    if [ x$feature_platform_search_hint = xy ]; then
      search -no-floppy -fs-uuid -set=root -hint-bios=hd0,gpt6
   -hint-efi=hd0,gpt6 -hint-baremetal=ahci0,gpt6
   35a31811-d3e1-4998-a239-e62929a8cca5
    else
      search -no-floppy -fs-uuid -set=root
   35a31811-d3e1-4998-a239-e62929a8cca5
    fi
    echo  'Loading Linux 4.13.0-rc2-ima-custom+ ...'
    linux  /boot/vmlinuz-4.13.0-rc2-ima-custom+
   root=UUID=35a31811-d3e1-4998-a239-e62929a8cca5 ro ima_tcb
   ima_template=ima-cont-id ima_cache1=false ima_cache2=false
   quiet splash $vt_handoff
    echo  'Loading initial ramdisk ...'
    initrd  /boot/initrd.img-4.13.0-rc2-ima-custom+
  }
```

Figure 5.1. Extract of grub2 kernel boot file

Save, reboot and select the new kernel.

5.2.3 Activating an IMA custom policy

In some cases it is not enough to stick with the standard ima policy. For example can be needed to measure all the files, even the one opened in read mode. It is possible to define a new custom IMA policy just by adding or removing IMA actions into the file */etc/ima/ima-policy*, and then save and reboot the machine IMA. During boot time this file will be copied under the directory *<securityfs>/ima/policy* and the new policy will be activated.

However, since the syntax offered by IMA is quite basic, it is possible to write the new policy rules by using the syntax defined by LSM. This is a framework that gives support to the security module inside the kernel and gives access control on files. The principal module that use this framework is *SELinux* (*Security-Enhanced Linux*). This module defines a new concept of *SELinux* user that is completely different from the standard user. In fact a *SELinux* user can be associated with many linux normal users. The *SELinux* user defines the access privileges that it can have on certain file *label*. A *label* is a tag that can be associated as an additional information to a standard file. By the default some users are defined inside *SELinux*:

- *user_u* with role *user_r* used for linux account with standard privileges
- *sysadmn_u* with role *sysadm_r* used for administrative linux account
- *staff_u* with role *staff_r* and *sysadm_r* used for linux account with standard privileges that needs also to do privileges operations

Keeping that in mind it is possible to rewrite a new IMA policy for this project:

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

This new policy is written using the basic syntax of IMA and will instruct it to measure only the executable file (i.e. the ones that has the *MAY_EXEC* privileges) that are loaded (*BPRM_CHECK*) or mapped into memory (*FILE_MMAP*). Using this policy we guarentee to have inside the log only relevant measure for the Remote Attestation process.

5.3 Docker Configuration

This project uses as virtualization engine docker. So now we will briefly discuss how to correctly install and configure it. All the steps described below are done under Ubuntu 16.04 LTS.

5.3.1 Installation

Before starting the installation process, make sure that the reference installed on system are updated:

```
sudo apt-get update
```

Install the docker project:

```
sudo apt-get install docker.io
```

Now it's time to start the docker daemon:

```
sudo service docker start
```

To test that all it's ok we can start a container:

```
sudo docker -it ubuntu
```

This command will go into the docker registry and search for an image called *ubuntu*. If it is not available offline, it will be downloaded and than started in interactive mode (*-it* flag). If all it's ok you should be now inside your first container!

5.3.2 Enabling DeviceMapper

Before starting to work with the project, it is necessary to set *DeviceMapper* as the default storage driver. Docker support a wide variety of storage driver, and the one used by default varies among linux distribution. At the time of writing, the default storage driver for Ubuntu is *overlay2*. So go ahead and open up the docker configuration file:

```
sudo gedit /etc/default/docker
```

add the parameter `--storage-driver=devicemapper` under the key *DOCKER_OPTS*. Once done that, save the file and restart the docker daemon:

```
sudo service docker stop
sudo service docker start
```

To check that docker now runs on *DeviceMapper* use the command:

```
sudo docker info
```

Under the *GraphDriver* section it must show something like:

```
Storage Driver: devicemapper
  Pool Name: docker-8:6-4853215-pool
  Pool Blocksize: 65.54 kB
  Base Device Size: 10.74 GB
  Backing Filesystem: ext4
  Data file: /dev/loop0
  Metadata file: /dev/loop1
  Data Space Used: 3.848 GB
  Data Space Total: 107.4 GB
  Data Space Available: 56.53 GB
  Metadata Space Used: 5.063 MB
  Metadata Space Total: 2.147 GB
  Metadata Space Available: 2.142 GB
  Thin Pool Minimum Free Space: 10.74 GB
  Udev Sync Supported: true
  Deferred Removal Enabled: false
  Deferred Deletion Enabled: false
  Deferred Deleted Device Count: 0
  Data loop file: /var/lib/docker/devicemapper/devicemapper/data
  Metadata loop file:
   /var/lib/docker/devicemapper/devicemapper/metadata
  Library Version: 1.02.110 (2015-10-30)
```

Figure 5.2. Extract of the output of command `docker info`

5.3.3 Patching and Compiling the Docker CLI

In order to retrieve with ease some internal detail of docker, a new docker CLI command has been developed. This raise the need to compile a custom version of the docker CLI [19]. To start off, clone the git repository of the docker CLI:

```
mkdir docker_cli
cd docker_cli
git clone https://github.com/docker/cli.git .
```

Patch the source code with the patch provided:

```
git am --signoff < docker.patch
```

It is possible to compile directly the docker CLI, but this will require that all dependencies are correctly installed on the system. To simplify a bit the process, we can use a method called *Docker Inception*: it will start a new container with all the required dependencies already installed, compile the source code inside the container and the final result will be than copied back to the *build* folder. The only requirement for this method is to have the docker daemon up and running and, only for the first time, an active internet connection. The latter is required since the

compilation container will be downloaded from the docker repos. If you are not sure that the docker daemon is active, just type:

```
sudo service docker start
```

We can now start the compilation:

```
sudo make -f docker.Makefile rabinary
```

After the compilation process finish correctly, the executable file will be placed under the *build* folder. To install it on the system type:

```
sudo cp ./build/docker-linux-amd64 /usr/bin/docker
```

In order to verify the correctness, check that a new command *raInfo* is available in docker CLI. To do that use:

```
sudo docker --help
```

and check the presence of *raInfo* in the command list.

5.4 OAT Appraiser configuration

The Appraiser has a central role in the OAT architecture. In this section we will go through the installation and configuration process for the appraiser.

First of all create a directory and clone the repository in it:

```
mkdir OAT
cd OAT
git clone https://github.com/SECURED-FP7/secured-verifier.git .
```

Apply the provided patch using the command:

```
git am --signoff < oat.patch
```

Map the IP address of the verifier under the file */etc/hosts* and change the name of the local host to "*verifier*":

```
echo -e 'xxx.xxx.xxx.xxx \t verifier' >> /etc/hosts
echo 'verifier' > /etc/hostname
```

where *xxx.xxx.xxx.xxx* is the ip address of the verifier (you can see it by using the command *ifconfig*).

Now it's time to compile the source code by generating an RPM file ready to be installed into the system:

```
cd ABSPATH/verifier/OpenAttestation/Source
sh distribute_jar_package.sh
cd ABSPATH/verifier/OpenAttestation/Installer
sh rpm.sh -s ABSPATH/verifier/OpenAttestation/Source
```

where *ABSPATH* is the absolute path to the *OAT* folder (i.e. if the folder *OAT* was created under the home directory, *ABSPATH* will be */home/fabio/OAT*). Once the compilation process is completed a RPM file will be generated under */root/rpmbuild/RPMS/x86_64/OAT-Appraiser-Base-OATapp-1.0.0-2.e17.centos.x86_64.rpm*. Before installing the RPM package be sure that the *mariadb* service is started, because, during that phase, the database used for OAT will be created.

```
sudo systemctl start mariadb
sudo systemctl enable mariadb
cd /root/rpmbuild/RPMS/x86_64
yum localinstall OAT-Appraiser-*.rpm
systemctl daemon-reload
```

Now that the Appraiser is correctly installed, it's time to configure it: First of all we need to generate the certificate that will be used to access the OpenAttestation Services:

```
cd ABSPATH/verifier/OpenAttestation/CommandTool
sh oat_cert -h verifier
```

Configure OpenAttestation:

```
sh configure_oat.sh selfName attestorName attestorIP PCR0_value
   OSname RApath DatabaseIP CertDigest
```

This command configure the Appraiser for executing the attestation of a particular host. The meaning of the parameters are the below:

- **selfName**: hostname of the verifier machine, in our case it is *verifier*
- **attestorName**: hostname of the machine to be attested
- **attestorIP**: IP address of the machine to be attested
- **PCR0_value**: value of the PCR0 registry of the TPM inside the machine to be attested. It is used when the command *VALIDATE_PCR* is issued
- **OSname**: Linux distribution of the machine to be attested
- **RApath**: path to the *ra_verifier.py*, in our case *ABSPATH/ra_verifier.py*
- **DatabaseIP**: IP address of the database containing the *whitelist*. In our case it is on the same machine of the Appraiser, so *localhost* will be good
- **CertDigest**: value of the SHA1 digest calculate on the file containing one certificate generated by the attested host. It is used when an analysis of type "cert-check" is issued

Add to the OAT database a new analysis type inside the *analysis_type* table, using *load-time+cont-check* as *name* and for the other fields the same values used in the other rows. Now it's time to open the port necessaries for receiving the Integrity Report:

```
sudo firewall-cmd --permanent --add-port=80/tcp
sudo firewall-cmd --add-port=80/tcp
sudo firewall-cmd --permanent --add-port=8443/tcp
sudo firewall-cmd --add-port=8443/tcp
```

Modify the file */etc/oat-appraiser/OAT.properties*, by uncommenting the properties `IR_DIR, IR_DIGEST_METHOD, SCALABILITY` and `DISCARD_IDENTICAL_IR`. Now it's time to test the verifier by opening up the page `http://verifier/OAT/alerts.php` in the browser. In this page it is possible to have a quick lookup of the IR received and memorized by the Appraiser and some other useful information.

Now it's time to install the reference database used for the integrity check of the IMA measure received into the IR:

```
cd ABSPATH/verifier/db/install
tar -xvzf apache-cassandra-1.2.19-bin.tar.gz
./install_cassandra_libs.sh
   ABSPATH/verifier/db/install/apache-cassandra-1.2.19
cd ABSPATH/verifier/db/install/apache-cassandra-1.2.19/bin
./cassandra > /dev/null
./cassandra-cli -h localhost -f
   ABSPATH/verifier/db/install/cassandra/schema/cassandra-
   schema-common.txt
./cassandra-cli -h localhost -f
   ABSPATH/verifier/db/install/cassandra/schema/cassandra-
   schema-rpm.txt
```

Copy the configuration file of the database into the directory */etc/ra*:

```
mkdir /etc/ra
cd /etc/ra
cp ABSPATH/verifier/db/conf/pkgs_download_list.conf .
cp ABSPATH/verifier/db/conf/ra.conf.sample .
```

Now open up the file *ra.conf* and substitute all the occurrencies of the word *RABASEDIR* with *ABSPATH/verifier*, remembering that *ABSPATH* is the absolute path to the OAT folder.

It's time to update all the packages with the command:

```
mkdir ABSPATH/verifier/Packages
cd ABSPATH/verifier/db/scripts
sh update_pkgs.sh
```

It is possible to test the corretness of the database by generating an *Integrity Verification Request* by issuing the command:

```
cd ABSPATH/verifier/v2
./ra_verifier.py -i
   ABSPATH/verifier/db/measurements/ascii_runtime_measurements -q
   CentOS7 -a "load-time,l_req=14|>=" -v -H localhost
```

If all is confiugred correctly, you should see something like:

```
Info: 0 (0/0)
0 (0/0)
0
0

Info: 0.00324
0.31203
0.08213
0.022021
0
0.34367
```

Figure 5.3. Extract of the output of the *ra_verifier.py* script

5.5 OAT HostAgent configuration

This section describes the required steps necessary to correctly install and configure an OAT HostAgent. The steps are referred to a machine with Linux Ubuntu 16.04 LTS. This machine require an active TPM that, in certain situations, needs to be activated from BIOS. For debugging purpose, or if a real TPM is not present on the machine, it is possible to use a virtual TPM as described in subsection 5.5.1.

First of all map inside the file */etc/hosts* the verifier hostname and IP address:

```
echo 'xxx.xxx.xxx.xxx \t verifier' >> /etc/hosts
```

where xxx.xxx.xxx.xxx is the IP address of the verifier. Start the service for the TPM:

```
sudo systemctl start tcsd
sudo systemctl enable tcsd
```

Download from the verifier the file *ClientInstallForLinux.zip*:

```
wget http://verifier/ClientInstallForLinux.zip
unzip ClientInstallForLinux.zip
```

It is possible to configure the file *OAT.properties* with the value of the properties desired. Install and register the host in the verifier using the command:

```
cd ClientInstallForLinux
sudo sh genera-install.sh
```

It is now possible to start the OATClient service:

```
sudo service OATClient start
```

The OATClient service will log his activities on the file */var/log/OAT.log*.

5.5.1 vTPM

As we have seen, it is mandatory for the OAT HostAgent to have an active TPM, . However, for different reasons, can be useful to use a virtual TPM. In this subsection are reported the instruction for installing and configuring one of the many vTPM [20] available today. First of all retrieve the source code of the vTPM, the one that we are going to use can be download with the command:

```
mkdir vTPM
cd vTPM
wget
   https://netix.dl.sourceforge.net/project/ibmswtpm/tpm4769tar.gz
   .
```

Then it's time to compile:

```
cd vTPM
make
```

Prepare the environment by creating the storage folder and setting the variables:

```
mkdir storage
sudo export TPM_PORT=2322
sudo export TPM_PATH=TPMABSPATH/storage/
sudo export TPM_SERVER_PORT=2322
sudo export TPM_SERVER_NAME=localhost
sudo export TCSD_TCP_DEVICE_PORT=2322
```

where *TPMABSPATH* is the absolute path to the *vTPM* folder created before.

Start the vTPM:

```
./tpm/tpmserver
./libtpm/utils/tpmbios //Always

/** Use this commands only the first time the vTPM is started **/
./libtpm/utils/createek
./libtpm/utils/nv_definespace -in ffffffff -sz 0
```

Now start tcsd by telling it to use the http protocol to connect to the vTPM:

```
sudo /usr/sbin/tcsd -e -f
```

Now the vTPM is up and running and it is possible to use it like a regular TPM.

It is highly suggested to use a regular TPM for production or critical environment, and use the vTPM only for testing purpose.

Chapter 6

Programmer manual

In the previous chapter we have discussed the architecture and the technical choice made in order to be able to do the remote attestation. In this chapter we are going to see in detail how the project has been implemented. All the code developed and here presented, is a modification of existing projects. In particular in section 6.1 we are going to see the changes made to docker, in section 6.2 the changes made to the IMA module of the linux kernel and in section 6.3 the changes made to the OAT HostAgent.

6.1 Docker changes

As we have discussed in chapter 4 a new command has been added to the docker CLI. All the code produced for docker has been developed using the GO programming language. First of all the new command has been developed inside the file */cli/command/container/ra_info.go*. The command has been developed as follow:

```
// +build rabinary

package container

import (
        "fmt"
        "strings"
        "os/exec"

        "github.com/docker/cli/cli"
        "github.com/docker/cli/cli/command"
        "github.com/spf13/cobra"
        "golang.org/x/net/context"
        "github.com/docker/docker/api/types"
)
```

```
type raInfoOptions struct {
        time        int
        timeChanged bool

        containers []string
}

// NewRaInfoCommand creates a new cobra.Command for docker stop
func NewRaInfoCommand(dockerCli *command.DockerCli)
   *cobra.Command {
        var opts raInfoOptions

        cmd := &cobra.Command{
                Use:   "raInfo [OPTIONS] CONTAINER
   [CONTAINER...]",
                Short: "Get info for Ra for one or more running
   containers",
                Args:  cli.RequiresMinArgs(0),
                RunE: func(cmd *cobra.Command, args []string)
   error {
                        opts.containers = args
                        opts.timeChanged =
   cmd.Flags().Changed("time")
                        return runRaInfo(dockerCli, &opts)
                },
        }

        flags := cmd.Flags()
        flags.IntVarP(&opts.time, "time", "t", 10, "Seconds to
   wait for stop before killing it")
        return cmd
}


func runRaInfo(dockerCli *command.DockerCli, opts *raInfoOptions)
   error {

        ctx := context.Background()

        options := &types.ContainerListOptions{
                Quiet: true,
        }

        var (
                cmdOut []byte
                err    error
        )
```

```
        cmdName := "lsblk"
        cmdArgs := []string{}

        if cmdOut, err = exec.Command(cmdName,
    cmdArgs...).Output(); err != nil {
                fmt.Fprintln(dockerCli.Out(), "There was an error
    running lslk command: ", err)
                return nil
        }

        out := string(cmdOut)
        //fmt.Fprintln(dockerCli.Out(), out)


        var errs []string

        containers, err := dockerCli.Client().ContainerList(ctx,
    *options)

        if err != nil{
                errs = append(errs, err.Error())
        }

        for _, container := range containers{
                c, err :=
    dockerCli.Client().ContainerInspect(ctx, container.ID)
                if err != nil{
                        errs = append(errs, err.Error())
                        continue
                }

                data := c.GraphDriver.Data["DeviceName"]

                if len(data)<0 {
                        errs = append(errs, "DeviceMapper not in
    use")
                        continue
                }

                fmt.Fprintln(dockerCli.Out(), container.ID[0:12]
    + " " + strings.Split(strings.Split(out, data)[1][1:], " ")[0])
        }

        return nil
}
```

Figure 6.1. *ra_info.go* source code

In particular:

- the first line, `// +build rabinary`, is a build tag that allows for conditional compilation. This tag tells the golang compiler to include (`+`) this file only if the tag rabinary is passed to the compiler

- the struct `raInfoOptions` defines the accepted input parameters of the command. `containers []string` is an array of string containing all the container's name / id to work with

- the function `NewRaInfoCommand` define the new command, with the hint for the `--help` flag of the CLI

- the function `runRaInfo` is the actual implementation of the new command. The input of this function is the command itself and the options passed to the command. First of all, in order to retrieve the mapping devId - DeviceName, the command `lsblk` is executed. Then for each container passed to the command, the internal function `ContainerInspect()` is called in order to retrieve low level information of the container. If the call to this function is successful, then the mapping devId - containerId is printed to the screen as described in chapter 4

This new command is then declared inside the file */cli/command/commands/commands.go* by adding the line:

```
...
hide(container.NewRaInfoCommand(dockerCli)),
...
```

Figure 6.2. Extract of *commands.go* source code

In order to be able to conditionally compile the project, some changes has been done also to the compilation infrastructure. In particular a new target called `rabinary` has been added to the file */docker.Makefile*:

```
...
#build executable for linux Remote Attestation
rabinary: build_docker_image
        docker run --rm $(ENVVARS) $(MOUNTS)
   $(DEV_DOCKER_IMAGE_NAME) make rabinary
...
```

Figure 6.3. Extract of *docker.Makefile* source code

We will now run a new development container and pass to it the command `make rabinary`.

Also a new build script has been added under */scripts/build/rabinary*:

```
#!/usr/bin/env bash
#
# Build a static binary for the host OS/ARCH
#

set -eu -o pipefail

source ./scripts/build/.variables

echo "Building statically linked $TARGET for remote attestation"
export CGO_ENABLED=0
go build -tags rabinary -o "${TARGET}" --ldflags "${LDFLAGS}"
   "${SOURCE}"

ln -sf "$(basename ${TARGET})" build/docker
```

Figure 6.4. *rabinary* source code

This script will be called by the makefile inside the development container. It invokes the golang compiler with the tag `rabinary` in order to include all the changes that has been made for the project.

6.2 IMA changes

As discussed in chapter 4, during the development of the project, we had the need to disable all the ima caches in order to be able to capture all the file loaded into memory, even if they are already measured. This behaviour cannot be modified without changing the source code of the IMA module.

The first changes are done inside the file */security/integrity/ima/ima.h* where two new flags has been defined:

```
...
extern int ima_cache1_enabled;
extern int ima_cache2_enabled;
...
```

Figure 6.5. Extract of *ima.h* source code

This new flags are correctly setted at bootstrap time by two `__setup` function defined inside the file */security/integrity/ima/ima_init.c*:

```
...
// Disable cache 1 and cache 2
static int __init ima_cache1_setup(char *str)
{
        if(strncmp(str, "false", 5)==0){
                printk("Disabling Cache1");
                ima_cache1_enabled = 0;
        }else{
                ima_cache1_enabled = 1;
        }

        return 1;
}
__setup("ima_cache1=", ima_cache1_setup);

static int __init ima_cache2_setup(char *str)
{
        if(strncmp(str, "false", 5)==0){
                printk("Disabling Cache2");
                ima_cache2_enabled = 0;
        }else{
                ima_cache1_enabled = 1;
        }

        return 1;
}
__setup("ima_cache2=", ima_cache2_setup);
```

Figure 6.6. Extract of *ima_init.c* source code

If the parameters `ima_cache1` and `ima_cache2` are passed as kernel boot parameters, this two functions are executed in order to correctly set to `enabled` or `disabled` the corresponding cache. If this parameters are not passed, then the two flags are initializated at `1` meaning that both caches are enabled, and therefore maintaining the standard behaviour.

The first cache consists of checking the ima *inode* information associated to the file to see if the file has already been measured by ima and if the file has changed in the meantime (this will work only if the filesystem has been mounted with `-i` flag, meaning that *iversion* is enabled). This cache has been bypassed by resetting the flag `measured_pcrs` to 0, into the ima inode information struct, after all successful measurements, inside the file */security/integrity/ima/ima_main.c*:

```
...
int ima_cache1_enabled = 1;
int ima_cache2_enabled = 1;
...
//Flush Inode For next Measurement if cache1 is disabled
if(ima_cache1_enabled == 0)
        iint->measured_pcrs = 0;
...
```

Figure 6.7. Extract of *ima_main.c* source code

Also, in this file, the two flags are initialized to 1, so that both caches will remains enabled if none of the two kernel boot parameters are specified.

The second cache consists in a lookup table where all the entries already printed in the log file resides. In order to bypass this limitation we skipped this lookup if the *cache2* is disabled. This has been done by modifying the function `ima_add_template_entry()` inside the file */security/integrity/ima/ima_queue.c* as follow:

```
...
mutex_lock(&ima_extend_list_mutex);
if (!violation) {
        memcpy(digest, entry->digest, sizeof(digest));
        if (ima_cache2_enabled==1 &&
   ima_lookup_digest_entry(digest, entry->pcr)) {
                audit_cause = "hash_exists";
                result = -EEXIST;
                goto out;
        }
}
result = ima_add_digest_entry(entry, 1);
...
```

Figure 6.8. Extract of *ima_queue.c* source code

now the function `ima_lookup_digest_entry()` will be called only if the *cache2* is enabled.

6.3 OAT HostAgent changes

In order to overcome the recent changes to the *DeviceMapper* driver in docker, and to continue to use this project with the latest version of docker, some changes are needed to the OAT Framework. In particular they are focused on the OAT HostAgent, and how it creates the mapping *devId* - *containerId* when a new IR request arrive. As discussed in chapter 4 two version of the same functions have been developed. The v1 is a bit slower but works with the standard version of docker (so no needs for recompilation). The v2 is way faster but requires the new command, developed for this project, *raInfo* to be available in the docker CLI.

6.3.1 v1 changes

All the changes in the OAT Framework are limited at the rewriting of the Java function `retrieveMapDmContainers()` in the file */OpenAttestation/Source/HisClient/src/gov/niarl/his/StandaloneHIS.java*:

```
...
/**
 * Retrieve a map indicating the association between DeviceMapper
 * virtual device numbers and the relative Docker container Id
 *
 * @return A map containing the association
 */
```

```
public Map<String, String> retrieveMapDmContainers() {
        Map<String, String> mappings = new HashMap();

        try{
                //Get the list of all container ids
                Process listOfContainerIds =
   Runtime.getRuntime().exec("docker ps -q");
                String id;
                BufferedReader br = new BufferedReader(new
   InputStreamReader(listOfContainerIds.getInputStream()));
                //Temporary mapping between devicename and
   containerid
                Map<String, String> devname_contid = new
   HashMap();
                while((id=br.readLine()) != null){

                        //Get Device Name of this container
                        Process getDeviceName =
   Runtime.getRuntime().exec("docker inspect "+id);
                        long startTime1 =
   System.currentTimeMillis();
                        BufferedReader br2 = new
   BufferedReader(new
   InputStreamReader(getDeviceName.getInputStream()));
                        String out;
                        String deviceName;
                        while((out = br2.readLine()) !=null ){
                                long insideStop =
   System.currentTimeMillis();
                                writer.println(" -- Inside:
   "+(insideStop-startTime1));
                                if(out.contains("DeviceName")){
                                        deviceName = out.split(":
   ")[1].replaceAll("\"", "").replaceAll(",", "");

   devname_contid.put(deviceName, id);
                                        break;
                                }
                        }
                        getDeviceName.waitFor();
                        getDeviceName.destroy();
                        long stopTime1 =
   System.currentTimeMillis();
                        writer.println("- Process:
   "+(stopTime1-startTime1));
                }
                //listOfContainerIds.waitFor();
```

```
                listOfContainerIds.destroy();

                long lsblkTime = System.currentTimeMillis();
                writer.println("After lsblk:
    "+(lsblkTime-startTime));

                //Analyze output of lsblk command to retrieve the
    final mapping
                Process lsblk =
    Runtime.getRuntime().exec("lsblk");
                br = new BufferedReader(new
    InputStreamReader(lsblk.getInputStream()));
                String line;
                while((line = br.readLine()) != null){
                        if(line.contains("docker") &&
    line.contains("dm")){
                                String[] lineSplit =
    line.split("\\s+");
                                String deviceName =
    lineSplit[1].substring(2);

    mappings.put(devname_contid.get(deviceName), lineSplit[2]);
                        }
                }
                //lsblk.waitFor();
                lsblk.destroy();

                long stopTime = System.currentTimeMillis();
                long elapsedTime = stopTime - startTime;

                writer.println("Time: "+elapsedTime);
                writer.close();

        }
        catch (Exception e) {
                e.printStackTrace();
        }
        return mappings;
}
...
```

Figure 6.9. Extract of *StandaloneHIS.java* source code in the first version proposed

These are the steps necessary in order to retrieve the mapping *deviceId - containerId*:

- Retrieve all the ID of the currently running container. This can be done by using the command `docker ps -q`: this command prints all the running containers, and the `-q` flag tells to print only the container ID one per line. The command is run by using the function `exec()` of java, and the output is then collected into an `InputStreamReader`. Using the function `readLine()` it is then possible to read the output line by line. Note that, since the function `readLine()` of the `BufferedReader` is a blocking function no error due to timing are possible

- For each *containerId* execute the command `docker inspect containerId` in order to retrieve some low level information. In particular we are interested of the field `DeviceName`, available inside the `GraphDriver` node. This will lead us to have a preliminary mapping *containerId - DeviceName*

- Run the command `lsblk` in order to retrieve the mapping *deviceId-deviceName*

- Put togheter the two intermediate mapping using the deviceName as the foreign key. This gives us the final mapping *deviceId - containerId*

In this piece code it is possible to see how the timing, shown in chapter 4, are retrieved: the current timestamp is collected at the start of the function in a variable called `startTime`, then at the end of the function the new timestamp is collected inside the variable `stopTime` and the final execution time of the function is then calculated as `stopTime - startTime`.

6.3.2 v2 changes

In the v2 all the changes are limited to the function `retrieveMapDmContainers()` of the file */OpenAttestation/Source/HisClient/src/gov/niarl/his/StandaloneHIS.java*. In this second case, since all the mapping logic has been moved into the docker CLI, the function can be simplified a lot:

```
...
/**
 * Retrieve a map indicating the association between DeviceMapper
 * virtual device numbers and the relative Docker container Id
 *
 * @return A map containing the association
 */
public Map<String, String> retrieveMapDmContainers() {
        Map<String, String> mappings = new HashMap();

        try{

                PrintWriter writer = new
   PrintWriter(DEFAULT_HIS_PATH+"mappingMeasure.txt","UTF-8");
```

```
                long startTime = System.currentTimeMillis();

                Process mappingCommand =
    Runtime.getRuntime().exec("docker raInfo");
                BufferedReader br = new BufferedReader(new
    InputStreamReader(mappingCommand.getInputStream()));
                String lineMap;
                while((lineMap=br.readLine())!=null){
                        String[] lineSplit = lineMap.split(" ");
                        mappings.put(lineSplit[0], lineSplit[1]);
                }

                long stopTime = System.currentTimeMillis();
                long elapsedTime = stopTime - startTime;

                writer.println("Time: "+elapsedTime);
                writer.close();

        }
        catch (Exception e) {
                e.printStackTrace();
        }
        return mappings;
}
...
```

Figure 6.10. Extract of *StandaloneHIS.java* source code in the second version proposed

The only command that is executed is the new command `docker raInfo`: this prints all the mapping *devId - containerId*, one per line. So to retrieve the mappings it is enough to parse the output line by line and split each line by the space char.

Chapter 7

Software alternatives

In this thesis work we have presented an architecture that let us achieve the Remote Attestation process on *Docker*. However some other software architectures are available in order to bring this process on a container based virtualization. In this chapter we are going to analyze and discuss some alternatives focusing on the trusted computing features they made available. In particular in section 7.1 we are going to analyze the *Intel Cloud Integrity Technology* (CIT), in section 7.2 the *Container Linux OS* and its *Rocket* container engine made by Core Os and in the last section we are going to compare them with the solution proposed by this thesis work.

7.1 Intel CIT

The *Intel Cloud Integrity Technology* (CIT) [21] is a platform developed by Intel in order to provide a Remote Attestation solution for all the Cloud components. It is the successor of the *Open Attestation Toolkit* (OAT) presented in chapter 2 and it is able to provide Trusted Computing both on standard and light virtual machine. It can also be used to introduce a geolocation and tagging mechanism into the infrastructure. The platform is based on the Intel proprietary architecture called *Trusted Execution Technology* (TXT).

7.1.1 Architecture

The basic architecture of the *Intel CIT* platform is independent on the virtualization engine used. A graphical representation is available in fig 7.1

The keys components of the *Intel CIT* architecture are:

- **Attestation Server**: It is the equivalent of the Appraiser in the solution presented in this thesis. It has the role to perform Remote Attestation on all the cluster nodes by comparing the measurement done by the Intel TXT (and extended to the TPM) with the ones saved into the whitelist database
- **Key Broker Service**: It creates and retains all the encryption/decryption keys of the various images loaded by the container in the cluster. When a

decryption keys is requested by a node, it acts as a broker by verifying the host's integrity status and by deciding at whom to respond

- **Trust Director**: It is used for generating VM images and for encrypt them by using the keys provided by the Key Broker Service. It is also responsible to generate the trust policy for each VM
- **Trust Agent**: It is the client side application that reside on each node that needs to be attested. It maintains the ownership of the Trusted Platform Module and acts as a proxy for all the request sent to him
- **KMS Proxy**: It is used to manage the launch request of encrypted images

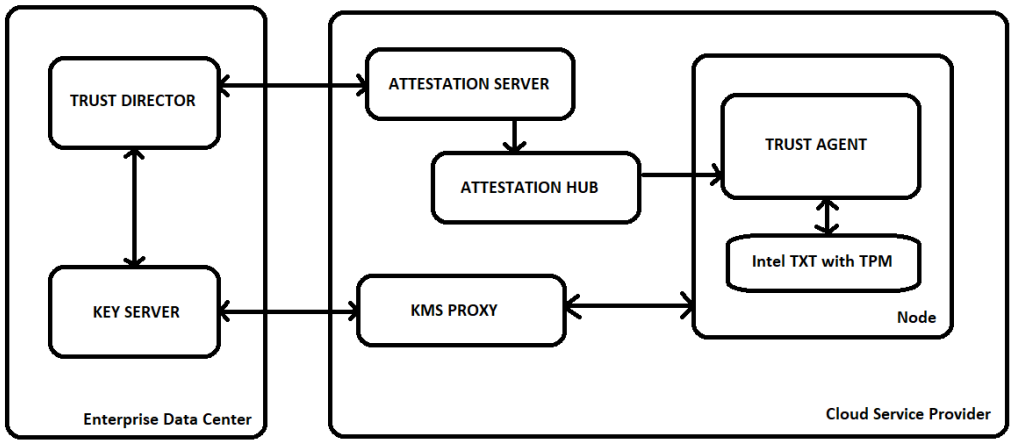

Figure 7.1. Basic architecture of the Intel Cloud Integrity Technology

With the *Intel CIT* architecture it is possible to encrypt VM images in order to enforce that an image can be run only by a verified host. When a host is required to run an encrypted VM image, it should follow this steps:

- The host request the key to the *KMS Proxy* and provide its *Attestation Identity Key* (AIK) taken from its TPM
- The *KMS Proxy* will check its integrity status with the *Attestation Server* using the AIK as the host Identity
- If the Verification is ok, the decryption key is wrapped with the AIK and is sent back to the host
- The host unwrap the key using its TPM and runs the decrypted image

The decryption key is then saved in memory and can be reused if a reboot of that VM is required. However, since it is stored in a volatile memory, after a host reboot, it is required to redo the procedure in order to retake the key.

7.1.2 Intel TXT

Intel Trusted Execution Technology (TXT) [22] is an Hardware architecture presented by Intel in order to validate platform trustworthiness during the boot and launch of software. In order to work the system must be equipped with a *Trusted Platform Module* (TPM) v1.2 on which *Intel TXT* can extends the measure done. It is used as the Root of Trust for the Intel Cloud Integrity Technology platform.

In order to work, it requires the following components:

- **Intel VT-d**: Intel proprietary on-chip Virtualization technology with support for direct I/O
- **TPM v1.2**: Security chip compatible with the standard v1.2 proposed by the Trusted Computing Group (TCG)
- **ACM enabled BIOS**: Bios with the support for Authenticate Code Module technology

The *Authenticated Code Module* (ACM) [22] is a code module that contains the procedure to prepare the system for *Intel TXT* and it is digitally signed by the chipset manufacturer. When establishing a new chain of trust, it is the first that is measured by the processor. When its signature and integrity are successfully validated, it will then proceed to validate the first module of the BIOS. All this measure are then extended to the PCR0 of the *Trusted Platform Module* (TPM) and they are considered the root of trust.

With the *Intel TXT* technology is possible to assign tags, such as geolocation information, to the different nodes (assets). The digitally signed set of asset tags compose the *Asset Tag Certificate* which is stored directly into the *Trusted Platform Module*. It is then used by the *Intel CIT* architecture to implement the *Boundary Control* and *Trusted Location* features or to define the type of workload that a node is able to do.

Boundary Control is a technology that lets the sys admin to force the processed data inside a particular location. For example it can be used to prevent that a particular VM image can be executed in a particular country or location, due to local law limitations. In case of hosts located in multiple locations, it is possible to force the *Intel CIT* to execute a particular workload on a trusted location. All this features are based on the *Asset* and *GeoTag* information available in the *Intel TXT* architecture.

7.1.3 Image Integrity

When executing a VM image that has been setup for *Image Integrity* some checks are done at load time. The first step is the decompression of the image. The second step is to measure all the files contained in the image. Each measure is than compared to the one saved in the *WhiteList DB* during the initial setup of the image. The behavior is then different based on the image integrity policy that has been selected:

- **Hash Only**: the image will be checked for integrity at load time and in case of failure it will be launched anyway but an alert is sent by the system

- **Hash and Enforce**: the image will be checked for integrity at load time and in case of failure the image will not be launched

Since the platform supports a wide variety of VM engines, the image signature feature is not directly available. However it is possible to activate it directly in the virtualization engine of choice.

7.2 Core Os

Core Os [23] is a company based in San Francisco with the aim to develop an enterprise Trusted Computing architecture for container based virtualization. In 2013 they released *Container Linux*, which is a virtualization based Operating System, and in 2014 they released a new container engine called *Rocket (rkt)*.

7.2.1 Container Linux

Container Linux [24] is a lightweight, virtualization based Operating System developed by Core Os. All the applications are required to run on different containers. It supports both docker and rocket as the container engine. It is designed in order to provide greater scalability, ease of use, automation and security to cluster deployments. All the various nodes are managed using a cluster manager called *Kubernetes*.

Kubernetes [25] is an open-source platform developed by *Google* in order to automatically manage a cluster of container nodes. It is structured with a Client-Server architecture. The basic unit is the *Pod* that corresponds to a set of containers that needs to run on the same location. All the nodes are managed by the Master by talking to the kubelet, that is an application that runs on the host and manage all the pods locally.

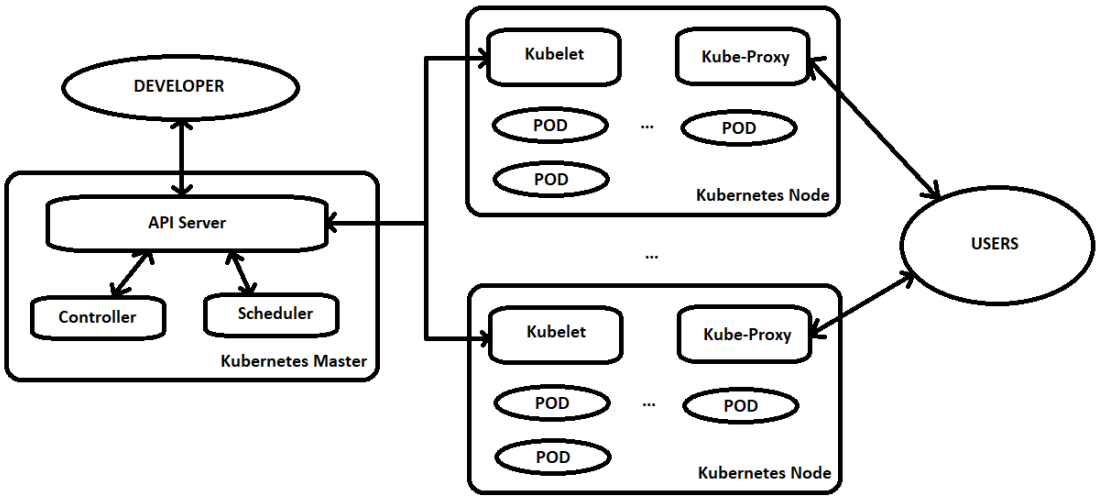

Figure 7.2. Basic architecture of Kubernetes

7.2.2 Rocket

Rocket (rkt) [26] is a container engine developed by *Core Os* with the aim to provide security and efficiency. Unlike docker that has a central deamon for managing all containers, rocket will run each container into a new Unix process. This allows rocket to achieve isolation by exploiting the standard Unix process isolation.

The basic unit in Rocket is the pod. A pod is a set of containers that needs to be run on the same location in order to exploit some host specific features. The pod startup process is divided into the following stages:

- **Stage 0**: It runs the rkt binary and do some preparation task like creating the filesystem for the pod or generating the pod UUID
- **Stage 1**: Read the Pod manifest and create the necessary isolation level
- **Stage 2**: Run the actual application chosen by the user

For each pod to be launched, it is possible to define on his manifest the appropriate isolation level. The available values are:

- **Fly**: It is the lowest possible isolation level. It is implemented as a simple chroot environment
- **Systemd/nspawn**: It is the medium isolation level. It is implemented using a combination of cgroups and namespaces functionality using systemd or systemdnspawn. This options correspond to the isolation level offered by Docker
- **kvm**: It is a full isolated environment implemented through the Kernel Virtualization Module (kvm) of the Unix kernel

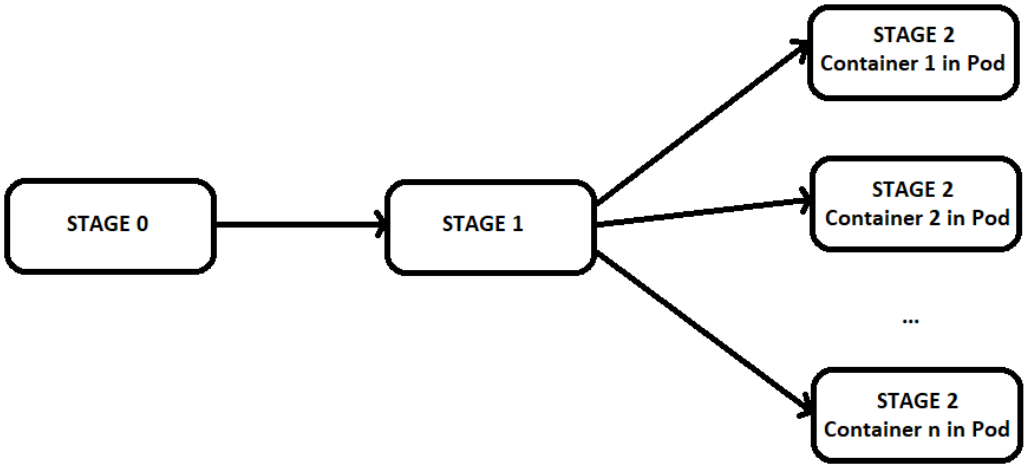

Figure 7.3. Stages of Rocket launch

By choosing the highest isolation level (kvm) Core Os will use a technology called *Intel Clear Containers*. It was initially developed by Intel with the *IntelVT* standard as a requirement, but now is supported also by Core Os in the rkt container engine with no hardware requirement. The basic idea is to optimize a standard hypervisor (qemu in the Intel version, kvm in Core Os) in order to reduce its memory footprint and startup time. This is done by optimizing both the kernel and the filesystem and by exploiting the direct access feature available in Unix kernel v4.0+. The result is a Virtual Machine that share all the isolation benefit of an hypervisor, but it is close to the startup time and memory utilization of a container.

7.2.3 Trusted Computing

Core Os provides Trusted Computing by using *Kubernetes*: a node is allowed to join the cluster if and only if its integrity status is verified. The verification of a node starts from the boot phase by verifying the integrity status of the hardware and of all the software up to the OS. Each VM image contains a signature that is checked with its configuration state by rkt when it must be loaded into memory.

The default image format used by Rocket is *Appc*, and it includes an image signature useful for checking the image integrity at load time. This feature is enabled by default and allows rkt to run only verified images. A similar feature is available with Docker by using the *Docker Content Trust* (DCT). No other integrity check mechanism are available.

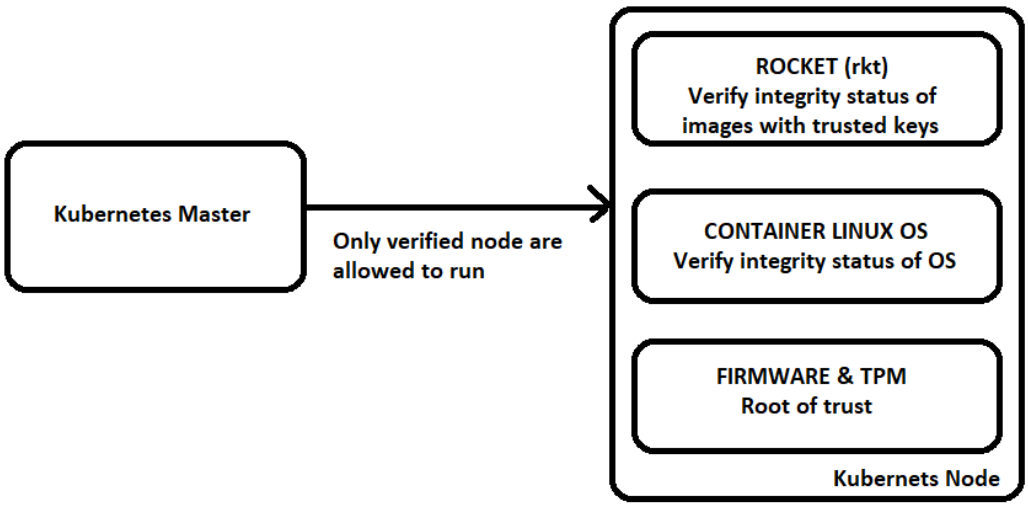

Figure 7.4. Trusted Computing Architecture in Core OS

7.3 Comparisons

In this section we are going to compare the two solutions analyzed in this chapter to the one proposed with by the thesis work. In particular we are going to compare them in terms of functionality and performance.

7.3.1 Features

All the solutions analyzed offers an *Image Integrity* feature. However both *Core Os* and *Intel CIT*, checks the integrity status only at load time. instead in the solution proposed in this thesis all the checks are done at run-time. With the SECURED project, if a container is attacked during the execution phase, it will be detected.

In all the solutions the image signature features is delegated to the container engine of choice. In particular with Docker is possible to use the *Docker Content Trust* mechanism, while Rocket provide a similar feature already enabled by default.

The *Intel CIT* platform supports a wide variety of Virtual Machine engines, ranging from the standard to the light ones. Instead *Core Os* supports only docker and rocket, while the SECURED project is based on docker.

No special hardware is required both for *Core Os* and *SECURED*, except for a *Trusted Platform Module v1.2*. Instead *Intel CIT* requires a compatible processor with the support of the *Intel VT-d* and *Intel TXT technology*.

For all the project the root of trust is based in hardware. However for *SECURED* and *Core Os* the chain of trust starts from the TPM while for *Intel CIT* starts from the *Intel TXT* hardware platform.

7.3.2 Performance

We can now move on by talking about performance. All the results presented in this section are obtained using the following machine:

- **Processor**: Intel Core i5 4670 @ 3.8 Ghz
- **RAM**: 8gb DDR3 @ 1600Mhz
- **Hdd**: 1Tb WD @ 6Gb/s
- **GPU**: Nvidia GTX 570

The first test takes into account the creation of a new container. From fig 7.5 it is possible to see that the solutions based on Docker are pretty similar and take less time in respect to the rkt one. Since Rocket runs every pod in a different Unix process, it has to load itself and all the dependencies every time a new container is loaded. This is not true for Docker, since all the container runs under the Docker daemon, that has preloaded all the basic dependencies.

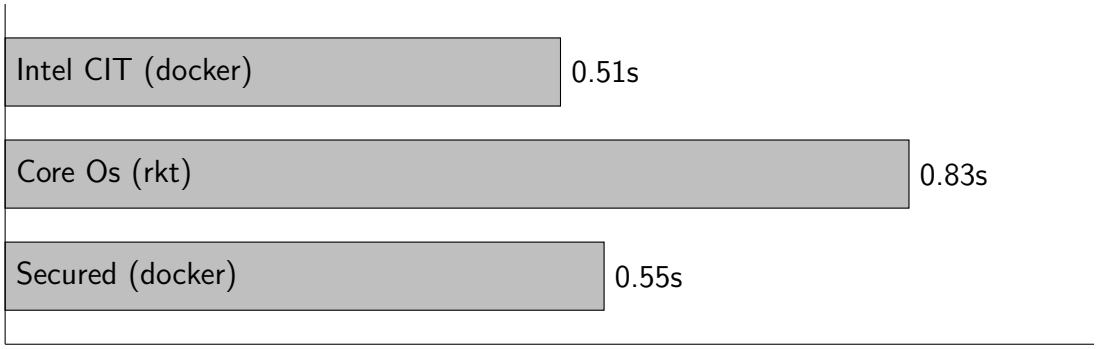

Figure 7.5. Average container startup time

The second test involves the time needed to create an *Integrity Report*. As it is possible to see in fig 7.6 all the proposed solutions are around 75ms, even if the architecture of the various project are quite different. It is important to note that the times are taken by using a network analyzer, like fiddler, on the Appraiser and calculate the time needed to receive back an *Integrity Report* when a new Attestation request happens. For this reason it must be taken into account also the serialization/deserialization time and the transfer time of the IR over the network.

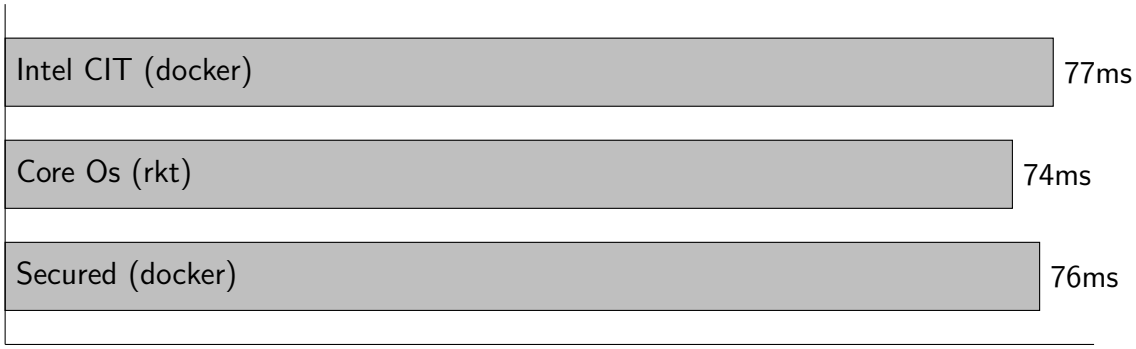

Figure 7.6. Average time to produce an IR with 5 active containers

Chapter 8

Results and conclusion

In this chapter are presented the results obtained during the development of this thesis work and the development of the solution proposed in chapter 4. Moreover will be presented some possible developments for the future and the conclusion of the thesis. In particular, in section 8.1 will be presented the results obtained during the development of the thesis work, in section 8.2 will be presented some possible future developments and in section 8.3 will be made the conclusions of the thesis.

8.1 Results

As we have seen, we've started our work with the need to make what already done by the TORSEC group to work also with the newest version of docker and IMA. Regarding IMA, no big changes where needed in this phase, but this is not true for docker. In fact from docker v1.10+ a lot of features has been introduced. In particular, in order to reduce the collision domain of container mappings, has been changed the way *Device Mapper* maps each container to the respective underlying virtual device. This project is heavily dependent on the mapping between the device id and the container id, that unfortunately with docker v1.10+ is broken. So we've started analyzing the docker source code in order to find how to achieve the same result and we've found that the device folder created by devicemapper for each container is named with a random hash. However this association is saved into memory and it is available by running the command `docker inspect containerId`. A screen of the interesting portion of this output can be seen in fig 8.1

With that information in mind, a new mapping procedure has been introduced. In particular it involves the following steps:

- List all the active containers by using the command `docker ps`
- For each container retrieve the *Virtual Device Name* by using the command `docker inspect containerId` and analyze the output of the command paying attention to the `GraphDriver` node
- Analyze the disk block status by using the command `lsblk`

- Retrieve the *deviceId* - *containerId* mapping by merging the information retrieved by the previous commands and by using the Device Name as a join key

```
''GraphDriver'':{
     ''Name'': ''devicemapper'',
     ''Data'':   {
                 ''DeviceId'': ''27'',
                 ''DeviceName'':
    ''docker-8:6-4853215-b665ee05b818394a5e98a294362cb955dc
                                  1a5211a412222ea840a627d72194bb'',
                 ''DeviceSize'': ''10737418240''
      }
},
```

Figure 8.1. GraphDriver node available in the output of the `docker inspect` command

The new mapping procedure works perfectly, however it introduces an unwanted latency in the creation of the *Integrity Report*. After a brief analysis on the created code, we've identified that this issue was created due to the presence of synchronous I/O parsing that is needed in order to retrieve all the information necessary to the mapping procedure.

Since the major part of the information needed to the mapping are already available on the docker *Command Line Interface* and the GO programming language [27] has a great integration with the linux sys call, we've decided to move the new mapping logic to the docker CLI by developing a new command, that has been called `raInfo`. It will output the mapping between each active *containerId* and *deviceId* by exploiting the internal raw level information available in the docker daemon. As it is possible to see in table 1, available in section 4.3, with this new technique the latency times were drastically reduced, returning in an acceptable range. It is important to note that this command is available only if the docker CLI has been compiled with the `buildRa` compilation tag. This feature has been done by exploiting the functionality provided by the *GOlang* compiler.

During further development of the project a new major problem arise. We've seen in chapter 4 that not all the file that meets the requirement set by the policy are measured. In order to resolve this problem we've start analyzing the source code available for the IMA module of the kernel. We've found that, in order to reduce the impact in term of performance losses of the module on the system, it uses two different caches mechanism to avoid to remeasure the same unmodified file. Unfortunately, for our project, this is a problem that needs to be solve. In fact we've the need that all the file that has been loaded to a container must be measured, even if that file has already been measured with another container. This feature is needed for the fact that the whitelist is different for each container.

The first cache is represented by and integrity structure attached to the Inode of each file. In this structure is available a flag, called `MEASURED`, that says if the file has already been measured for that *Inode* version. If that flag is true and the *Inode* version has not been changed, that IMA assumes that this file is not changed from the last measure, so no needs to remeasure it. The *Inode* version is changed every time a file is modified, but only if the filesystem has been mounted with the `-i_version` flag. The solution for disabling this first cache is to force the `MEASURED` flag to false every time a new measure is done. For clarification the Integrity structure attached to the Inode of the file is available in fig 8.2

```
/* integrity data associated with an inode */
struct integrity_iint_cache {
        struct rb_node rb_node;
        struct inode *inode;
        u64 version;
        unsigned long flags;
        unsigned long measured_pcrs;
        enum integrity_status ima_file_status:4;
        enum integrity_status ima_mmap_status:4;
        enum integrity_status ima_bprm_status:4;
        enum integrity_status ima_read_status:4;
        enum integrity_status evm_status:4;
        struct ima_digest_data *ima_hash;
};
```

Figure 8.2. Integrity data associated to the Inode of each file

The second cache is composed by an hashtable that stores all the measures already printed on the log file. In particular each time a new measure is done, IMA will check if the hash of the measure is available in the hash table and, in that case, it will discard the measure. The solution for disabling the second cache was to add a condition in the check of the hash table.

It is important to note that all the modification introduced to the IMA module of the kernel will not affect in any way all the functionality provided by it. In fact we've added two new kernel boot parameter, called respectively `IMA_CACHE1` and `IMA_CACHE2`, that says if the respectively cache has to be enabled or disabled. In particular if they are not set, IMA will stick to standard behaviour so that both caches are enabled.

8.2 Where to go next

A lot of work has been done during the development of the thesis. However some improvements can be done to the project in order to make it more secure, flexible

and efficient. In this section we are going to analyze some possible improvement that can be done in the future.

The first problem that needs to be addressed is a potential flaw in security. It is related to the fact that each measure in the *Integrity Report* is binded to a particular container by using a mapping *containerId* - *deviceId* and the way docker behaves when a stop command is issued. In particular when we start a container, *Device Mapper* will assign the first *deviceId* of the pool available to it. Instead when we stop it, the association is broken and this device id became again available, so that if a new container is started it can takes it. The problem is that, in the current solution, we are not tracking in anyway this change in the device Id association, so that if a new container will take a previously used deviceId, it will also inherit all the measure done on the old container on that device id, possibly resulting in a fail when an Integrity check is done in the new container. A graphical explanation of this problem is available in fig 8.3

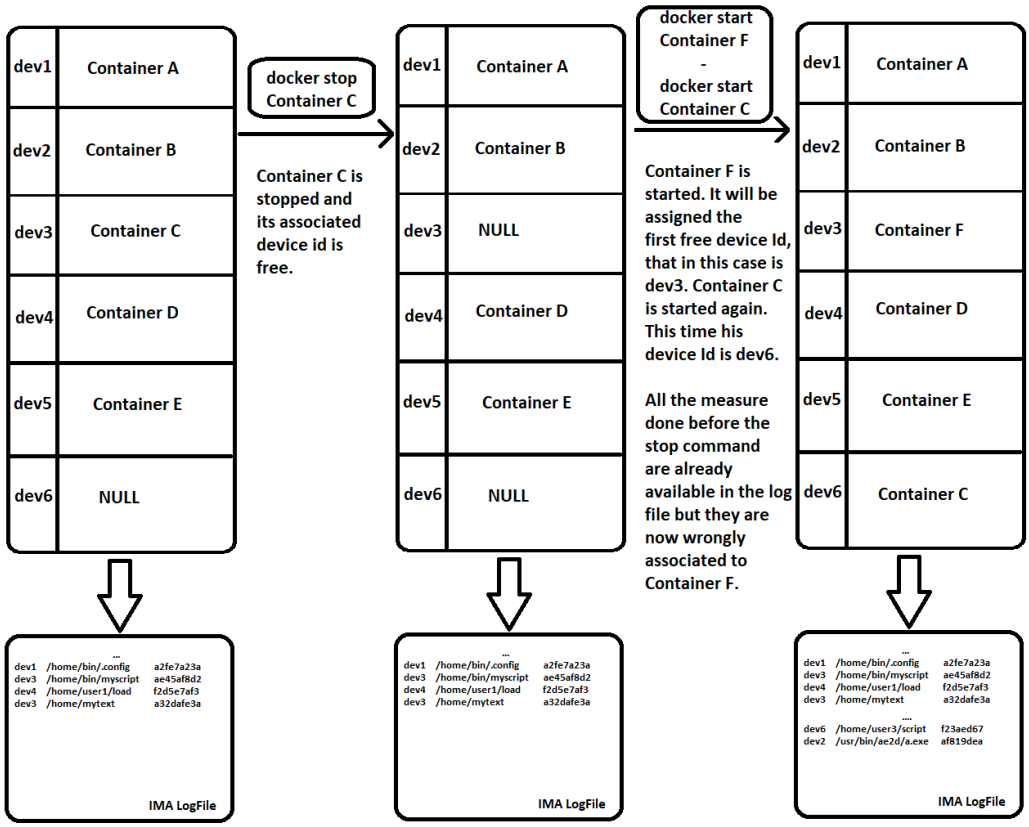

Figure 8.3. Graphical representation of the STOP command problem

One possible solution to that problem is to add a special row in the IMA measurement list in order to track the status change of a particular container, so that when the appraiser is going to check the integrity status of a container, it will consider only the newer measure in the IR, that are the one after the last `START` command. But since IMA is protective regarding his log file, it is not possible to manually add

a row in his log file, otherwise the integrity check of the *Integrity Report* created will fail. So for resolve this issue IMA needs to expose on a unix socket a set of API that can be used to work with the log file, and possibly add a new measurement line into it. However this solution will greatly increase the surface attack of IMA, so some precautions, like access privileges to the Unix socket, must be put in place in order to avoid attack exploiting this new feature.

For example one possible attack is represented in fig 8.4. In a normal situation if someone manages to run a bad behaving program or a script inside a container it will be detected since the signature of the file loaded is not available into the WhiteList database. However if the attacker manages also to write a fake `STOP` record inside the IMA measurement list exploiting the API provided by IMA itself, then all the measure before that line will not be considered, and the bad script will not be detected.

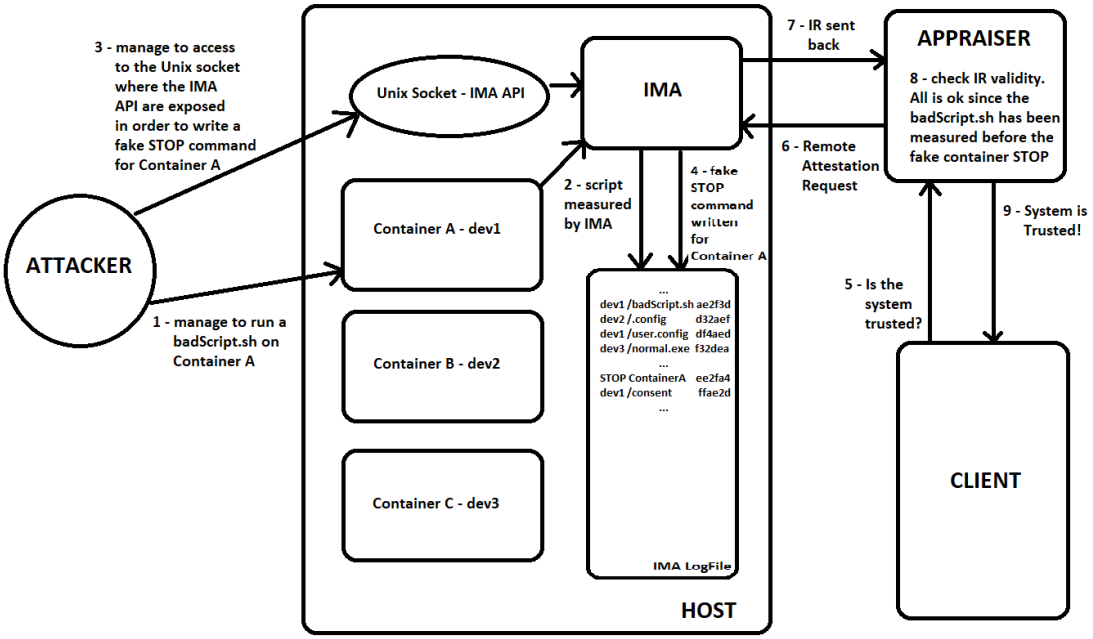

Figure 8.4. Possible attack on the solution proposed for the STOP command

The proposed solution make use of a modified version of IMA in order to resolve some security flaw. In particular two new kernel parameters has been added in order to deactivate, when needed, the cache putted in place by the system. Currently IMA is undergoing a rewriting of the module. The new IMA module will be available in the first quarter of the next year and it will embed the possibility to control each cache independently by simplying changing a configuration file. Also there will be more room to custom template, so if the project will follow this remake of IMA it can be avoided to patch and recompile the linux kernel for each node to be attested. This will be a great improvement for the usability and maintenance of the platform.

Right now the solution proposed is highly binded to the *DeviceMapper* storage driver. However there are other Storage Driver supported by Docker that offer better

performance and stability in respect to Device Mapper. In particular the future for Docker seems to go in the direction of *Overlay2fs* that gives better performance in respect to the "old" Device Mapper. In fact most of modern linux distro uses as default *overlay2fs* in docker. So one possible development is adding the support for that particular Storage Driver.

8.3 Conclusions

We start by focusing on how *Cloud Computing* has changed the way we access IT services and the needs to provide security in it. In particular we focused on the importance to establish trust on a particular node and how this can be done by using the methodologies introduced by the *Trusted Computing Group*. However the framework developed by the TCG aims to provide trust on a physical hardware machine, but nowadays most of the services are executed inside a *Virtual Machine*. In this way is possible to provide greater flexibility and scalability to the end user of the provider services. For this reason, the *Trusted Computing Group*, decided to change a little his specification in order to provide Trust also on *Virtual Machine*. This can be done by using an *Integrity Verificaton Proxy*, that is a middleware component that manages all the connection to the various *Virtual Machine* granting the Integrity status of them in respect of the criteria stated by the client at the moment of the connection. Since the IVP is a critical component it must be trusted, but this can be done by simplying using the standard *Remote Attestation* procedure already available.

In the last few years we have assisted to a rapid grow of a new virtualization technique called *Light Virtualization*. This one promises the same benefit of a regular *Virtual Machine* but without adding the overhead of a standard Virtualization technique. The aim of the thesis was to extend the support of the *Remote Attestation* process also to this type of Virtualization method.

In particular the thesis is inserted inside the context of the SECURED project, where it will help the development of the *Network Edge Device* function inside the architecture proposed by SECURED. Some parts of the development was already done by the TORSEC research group but changes were needed to makes it available to the newest docker version and to resolve some issues found during the analysis.

The main aim of the thesis was to finish with a functional framework that can be used inside the SECURED project in order to implement and trust the figure of the NED. The first step toward this aim was to take what already done by the TORSEC group and, after a brief analysis, makes it work with the newer version of docker and IMA. In order to do that, the work has started by analyzing in detail how devicemapper works and how docker internally maps each containers to the virtual device created. We have found that, unfortunately, there is no direct link between a container and its device mapper folder, however this mapping is saved as an internal information and can be partially retrieved by using the command `docker inspect`. With that information in mind a new mapping procedure has been created and the project was functional again for docker version 1.10+. A second version of this new mapping procedure has been done in order to make it faster. This second version

includes some modification to the docker CLI, and so I've had the opportunity to study a new programming language called GO.

After some developments a new major problem araised: not all the file loaded inside a container were measured by IMA. After a brief analysis we've discovered that this is done because IMA maintains two different caches mechanism. With this particular problem I've had the chance to analyze the code behind the IMA kernel module and to develop a kernel patch in order to add two new kernel boot parameter for disabling the two caches mentioned before.

Overall has been a very good experience that lets me study some technology that were new for me, like the OAT framework and the whole *Trusted Computing* technologies. Furthermore, in order to achieve the aim of the thesis, I had the chance to analyze the source code of docker and of the linux kernel and then to create some patches for it. In conclusion it has been a very pleasant experience that let me grows my knowledge in a particular area of cybersecurity, and at the end all the aim stated at the beginning of the thesis have been reached. I also hope that my thesis work will be useful in order to develop the SECURED project and build a safer cyber world!

Bibliography

[1] P.Mell, T.Grance, "The NIST Definition of Cloud Computing", September 2011, pp. 6-7, DOI 10.6028/NIST.SP.800-145,

[2] M.Eder, "Hypervisor-vs. Container-based Virtualization", WS 2015/2016: Seminar Future Internet, Technische Universität München (Germany), July 1, 2016, DOI 10.2313/NET-2016-07-1_01

[3] The SECURED project, https://www.secured-fp7.eu/

[4] D.Montero, M.Yannuzzi, A.Shaw, L.Jacquin, A.Pastor, R.Serral-Gracià, A.Lioy, F.Risso, C.Basile, R.Sassu, M.Nemirovsky, F.Ciaccia, M.Georgiades, S.Charalambides, J.Kuusijarvi, F.Bosco, "Virtualized Security at the Network Edge: A User-centric Approach", IEEE Communications Magazine, Vol. 53, No. 4, April 2015, pp. 176-186, DOI 10.1109/MCOM.2015.7081092

[5] The TORSEC research group, http://security.polito.it/

[6] R.Shirey, "Internet Security Glossary, Version 2", RFC-4949, August 2007, DOI 10.17487/RFC4949

[7] The Trusted Computing Group (TCG) website, http://www.trustedcomputinggroup.org/

[8] Trusted Platform Module (TPM) v1.2 specifications, https://trustedcomputinggroup.org/wp-content/uploads/PC_Client_TPM_PP_1.3_for_TPM_1.2_Level_2_V116.pdf

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

[10] The Open Attestation Toolkit (OAT) project, https://01.org/blogs/tlcounts/2014/openattestation-oat-project

[11] J.Schiffman, H.Vijayakumar, T.Jaeger, "Verifying System Integrity by Proxy", TRUST 2012: 5th International Conference on Trust and Trustworthy Computing, Vienna (Austria), June 13-15, 2012, pp. 179-201, DOI 10.1007/978-3-642-30921-2_11

[12] The Docker project, https://www.docker.com/

[13] The Docker Command Line reference guide, https://docs.docker.com/engine/reference/commandline/docker/

[14] Docker layer fs reference guide, https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/#images-and-layers

[15] Supported storage drivers in Docker, https://docs.docker.com/engine/userguide/storagedriver/

[16] Security features in Docker, https://docs.docker.com/engine/security/security/

[17] The Docker Content Trust (DCT), https://docs.docker.com/engine/security/trust/content_trust/
[18] How to compile a Kernel in Ubuntu, https://help.ubuntu.com/community/Kernel/Compile
[19] Contributing to Docker, https://github.com/moby/moby/blob/master/docs/contributing/set-up-dev-env.md
[20] The IBM virtual Trusted Platform Module (vTPM), http://ibmswtpm.sourceforge.net/
[21] The Intel Cloud Integrity Technology (CIT) product guide, https://github.com/opencit/opencit/wiki/Open-CIT-3.2.1-Product-Guide
[22] W.Futral, J.Greene, "Intel Trusted Execution Technology for Server Platforms", Springer, 2013, pp. 15-36, DOI 10.1007/978-1-4302-6149-0
[23] The Core Os official website, https://coreos.com/
[24] Container Linux OS official guide, https://coreos.com/os/docs/latest/
[25] Kubernetes official website, https://kubernetes.io/
[26] Rocket (rkt) documentation, https://coreos.com/rkt/docs/latest/
[27] The GO programming language, https://golang.org/