Integrity verification of Docker containers for a lightweight cloud environment

Marco De Benedictis[a,*], Antonio Lioy[a]

[a]*Politecnico di Torino, Dip. Automatica e Informatica, Corso Duca degli Abruzzi 24, 10129 Torino, Italy*

Abstract

Virtualisation techniques are growing in popularity and importance, given their application to server consolidation and to cloud computing. Remote Attestation is a well-known technique to assess the software integrity of a node. It works well with physical platforms, but not so well with virtual machines hosted in a full virtualisation environment (such as the Xen hypervisor or Kernel-based Virtual Machine) and it is simply not available for a lightweight virtualisation environment (such as Docker). On the contrary, the latter is increasingly used, especially in lightweight cloud platforms, because of its flexibility and limited overhead as compared to virtual machines. This paper presents a solution for security monitoring of a lightweight cloud infrastructure, which exploits Remote Attestation to verify the software integrity of cloud applications during their whole life-cycle. Our solution leverages mainstream tools and architectures, like the Linux Integrity Measurement Architecture, the OpenAttestation platform and the Docker container engine, making it practical and readily available in a real-world scenario. Compared to a standard Docker deployment, our solution enables run-time verification of container applications at the cost of a limited overhead.

Keywords: security management, cloud computing, trusted computing, remote attestation, lightweight virtualisation, integrity verification

[*]Corresponding author

Email addresses: `marco.debenedictis@polito.it` (Marco De Benedictis), `antonio.lioy@polito.it` (Antonio Lioy)

1. Introduction

Current ICT infrastructures benefit from modern technologies such as those of the cloud computing paradigm, wherein virtualisation is used to optimize the usage of hardware platforms and to simplify service management, leading to improved flexibility, availability, and reduced costs. We refer to these architectures as *softwarised* environments as they use commodity hardware customised via software. While virtualisation offers various advantages from the security point of view (such as isolation and sandboxing), at the same time it creates issues because services execute through a virtualisation manager, which in principle can alter their data and operations. It is therefore important to guarantee code and data integrity for services running in softwarised environments.

The Trusted Computing architecture, proposed by the *Trusted Computing Group* (TCG) [1], aims to provide strong protection against various software problems, such as malware, network attacks, and configuration errors. This is achieved by providing authentic evidence of the platform software state, with the help of a special hardware component which often takes the form of the *Trusted Platform Module* (TPM) chip. The *Remote Attestation* (RA) technique exploits this evidence to appraise the software integrity state of a node designed to the TCG specifications. RA can be used to evaluate the integrity of services running directly on a physical platform, because a direct link to the TPM is needed. In virtualised environments, the hypervisor can be a target of the attestation process because it is a service running in the host system and therefore it has direct access to all hardware components. However, the TPM-based integrity verification of physical compute nodes is not easily extensible to virtual instances. In fact, the virtualisation layer offered by the hypervisor may not export TPM resources to the *Virtual Machine* (VM). Even if the TPM resources are exported to the VM (as in the *Kernel-based Virtual Machine* (KVM) system that implements a pass-through driver for direct interaction between the VM and the hardware TPM), the TCG specification assumes a one-to-one relationship between the operating system and the TPM [2], which makes the chip unable to provide authentic evidence for all the VMs. This problem is the main obstacle to use RA effectively in a hypervisor-based virtualisation environment.

A lot of interest is currently being paid to lightweight virtualisation techniques, as they incur a lower performance penalty compared to full virtualisation because they create smaller and more agile execution environments, i.e. containers. According to a survey conducted by Datadog [3], 18.8% of its customers have adopted containers in their infrastructures by April 2017, which represents a 40% year-over-year growth in the number. Deutsche Telekom started experimenting with containers in cloud-based *Network Function Virtualisation* (NFV) environments since 2015 [4] and, more recently, AT&T has introduced containers in its enterprise cloud solution [5]. Lightweight virtualisation is especially important in target environments where the nodes have limited computational resources. Nonetheless, containers introduce security risks for the cloud platform as they provide less isolation than VMs, which may expose the host system to privilege escalation. Moreover, containers are based on *images* (i.e. archives which include binaries, libraries, and the root file-system of each virtual instance) whose management and distribution may introduce vulnerabilities for the target platform, as noted by the NIST [6].

Well-known container technologies propose solutions to ensure the integrity of images, but they do not cover the whole service lifetime, as the image and its internals may be changed at run-time by the host or by external attackers exploiting some vulnerability. For instance, an attacker that has gained a privileged access to a running container may compromise it by modifying service configurations and binaries, launching malicious scripts, or starting new processes, all without being detected by static verification techniques (since they are concerned only with load-time integrity).

To tackle this problem, we have developed a solution for software integrity attestation of a lightweight cloud environment at run-time, which covers both the host and the services in the containers. This solution is named *Docker Integrity Verification Engine* (DIVE), as it targets the Docker container engine

[7]. This is the de-facto standard for containerisation of applications on a Linux platform, as it is widely supported by vendors and production-oriented cloud environments. Nonetheless, our proposal is based on Linux kernel functionalities independent of the specific container runtime, hence it can be generalised to cover a wide spectrum of lightweight virtualisation technologies.

Our focus is only on the integrity of the services running in the host and in the containers themselves, but not on the privacy of the data or on the correctness of the computation, that are well-known problems of any virtualisation and cloud environment. However, note that if the software running in the host and the containers has been evaluated and erroneous or malicious behaviour is not present, then DIVE demonstrates that no other software component is executed in the environment, and this in turn supports trust in the correct behaviour of the node.

From the authors' perspective, this work represents a concrete proposal to provide run-time integrity evidence of services running inside virtualised instances with a direct link to the hardware TPM, in a reliable and scalable manner. On one hand, this solution enables the deployment of critical services in containers whose integrity state can be securely attested. On another hand, the ability to attest services in containers fulfils the requirement of integrity verification in lightweight cloud infrastructures. However, we do not claim to cover the full range of software attacks to virtualised instances, as the TPM-based attestation would not be able to detect in-memory manipulations of code or data. This limitation can be addressed by modern *Operating System* (OS) protection techniques such as *Address Space Layout Randomisation* (ASLR) [8]. In turn, DIVE can be used to attest if the host kernel has activated system hardening protections (such as ASLR) or not.

The rest of this paper is organised as follows: Section 2 presents some background about containers and Trusted Computing. Section 3 details our proposed integrity verification process and Section 4 presents the prototype of the proposed architecture. Section 5 discusses performance and scalability of the proposed architecture. Section 6 presents the state of the art and discusses our improvement over it. Section 7 summarises the contributions of this article.

2. Background

This section introduces the technological concepts at the base of our proposal. The lightweight virtualisation concept is briefly explained, along with a comparison among popular containerisation technologies. Then, the section covers the different aspects of the specific case study of our work, i.e. the Docker virtualisation engine. Moreover, an overview on the vulnerabilities of containers is exposed, as a means to justify the need for integrity verification in lightweight virtualisation. Afterwards, the section covers the Trusted Computing principles on integrity verification by means of a hardware-based root of trust. Finally, the Remote Attestation workflow is explained in detail, along with the run-time integrity measurement architecture.

2.1. Containers

Containers represent an OS-level virtualisation technique, also called *paenevirtualisation* [9] or lightweight virtualisation, where the OS kernel allows for multiple isolated user-space instances. Containers remove the overhead introduced by the hypervisor, which makes them smaller and faster to be started/stopped than hypervisor-based VMs [10].

2.1.1. Containerisation technologies

Lightweight virtualisation exploits the resource isolation features of the Linux kernel (such as cgroups [11], namespaces [12] and kernel capabilities [13]) and a union-capable file-system to allow independent containers to run within a single Linux instance. Docker [7] and Rkt [14] are well-known implementations of *process containers*, which build an isolated execution environment comprising a target application (e.g. a MySQL server) and its software dependencies (e.g. binaries, libraries). *Linux Containers* (LXC) [15] and LXD [16] represent alternative technologies that exploit the same kernel isolation features to offer a slightly different container run-time. These technologies are known as *machine containers* as they target isolation of multiple processes and services within a single container. They are easier to manage because they can be customised as traditional VMs, but they ultimately offer less flexibility, reuse, and composability (very important in highly dynamic virtualised environments such as the cloud ones). More recently, *Unikernels* [17] have been proposed as an alternative to Linux container technologies. These represent a radical shift from general-purpose virtual instances (such as VMs and machine containers) to fixed-purpose minimal images that run a single application, similarly to process containers. Unikernels are built into machine executables which embed the target application and its dependencies, and they can be run on an hypervisor or bare metal. Because of their minimalistic nature, they typically require limited system resources to run. In this work we focus on Docker containers, rather than unikernels, as they are exploited in production-oriented environments and they can benefit from orchestration systems that are particularly relevant to cloud platforms.

2.1.2. Docker Container Engine

Docker is an open-source project that uses containers to simplify the deployment of services by providing an additional layer of abstraction, the Docker Container Engine. It focuses on the *one process per container* methodology, which requires each virtualised instance to run a single application in foreground, meaning that the container lifetime would be equal to the target application runtime. A fundamental block in Docker is the image: containers themselves are launched from images, which can be considered as the “source code” for the containers. Images are built in a layered way, and layers are ordered and stored in a single file-system (Figure 1). At the base, there is a boot file-system, i.e. *bootfs*, which contains required data (e.g. the bootloader) to boot the container. The second level is composed by one or more root file-systems, called *rootfs* or base image. Each base image hosts a read-only layer of the OS image, comprising files and directories required for the overall functionality of the container. Every time a new container is launched, the Docker daemon constructs a read-

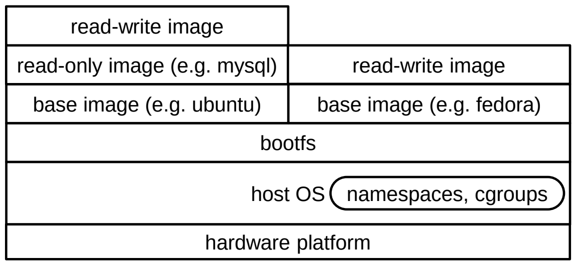


Figure 1: The Docker image hierarchy

write layer at the top of the read-only image hierarchy, in which all container processes are executed. Each image layer is managed by a storage driver that is responsible to manage the image layers and perform the copy-on-write operations. Storage drivers used by Docker are Aufs, Device Mapper, and Btrfs [18], where the former two are the most popular ones. Read-only image hierarchies can be shared among different containers, optimising the overall resource usage (e.g. disk and memory) occupied by containers. Docker provides both the container runtime itself, i.e. the Docker daemon, and additional components for the life-cycle management of containers, such as a local image repository and a declarative language to express container internal functionalities (e.g. base image, build-time commands, and exposed network ports).

2.1.3. Vulnerabilities of container technology

NIST [6] proposes a classification of the risks for the core components of containers regardless of the specific technology. First, container images can be exploited to carry untrusted software or to embed malware by an attacker. Moreover, the image creator may use software for whom a vulnerability is discovered at some time. In this case, unless the image is re-built in the container image service (e.g. the Docker local image repository), it will include the vulnerable software in all the containers instantiated from it.

Then, the image distribution process may introduce additional risks. Docker implements a registry service for image sharing, which can be exploited by multiple container engines running on different hosts to download images from a centralised entity. Insecure connection to the registry, weak server and client authentication, and lack of automatic monitoring of stale images may increase the risk for Man-in-the-Middle attacks and vulnerable software.

Moreover, container orchestration platforms may introduce security vulnerabilities in case they are not properly managed by the administrator. More specifically, the orchestrator should ensure that inter-container interaction is finely tuned, and workloads with different sensitivity levels are not mixed. Additionally, the trustworthiness of the orchestration platform should be verified.

Finally, vulnerabilities may be introduced in case the container engine itself (e.g. the Docker daemon) is poorly configured or it has been manipulated. In this regard, the container engine integrity should be verified along with its runtime configuration, to ensure that a malicious user cannot exploit the container engine itself to gain host privileges. Because of this, the host OS itself should be verified as well, and its attack surface should be limited as much as possible to prevent execution of vulnerable software and insecure configuration.

The risks can target the integrity of each container or its interaction with other containers or the host OS. To contrast these issues, an integrity verification mechanism should be in place to ensure the trustworthiness of the following entities: the runtime container image, which may carry both out-to-date and untrusted software, or even malware; the container engine itself, which may have been manipulated by an attacker to exploit a vulnerability; the underlying host OS, which may be manipulated in case an attacker manages to exploit an undetected container vulnerability to attack the OS. Docker natively offers a static and basic approach to integrity verification, as it can check the integrity of the image at load-time only. This is offered by *Docker Content Trust* (DCT) [19, 20], introduced in Docker version 1.8 to permit verification of the container image against the digital signature by its creator. Intel proposes a similar solution, named Trusted Docker Containers, to ensure that Docker images are not tampered prior to launch [21].

2.2. Trusted Computing technology for integrity verification

According to the TCG functional specification, a *Trusted Platform* (TP) [22] must implement three basic features: protected capabilities, integrity measurement, and integrity reporting. Protected capabilities are the commands that have exclusive permissions to operate on shielded locations, i.e. special regions of the platform where it is safe to store and operate on sensitive data. In the solution proposed by TCG, the shielded locations sit in the internal memory of the TPM and are called *Platform Configuration Registers* (PCRs). By design, each TPM has a minimum of 24 PCRs, numbered from 0 to 23, and each PCR is 20 bytes of size, i.e. the size of a SHA-1 digest. The TCG has produced several iterations of the TPM specifications, the latest being the release 2.0. This introduces separate PCR banks, wherein all registers are extended using the same hash algorithm, i.e. *Secure Hash Algorithm 1* (SHA-1). The integrity of the platform is defined as a set of metrics that identify software components, such as the OS, applications, and their configurations. For each component, a fingerprint acts as unique identifier and as proof of no modification; software components are *"measured"* via a *binary attestation* approach, i.e. by computing the cryptographic digests over their binaries on the file. Later, the measurement process uses protected capabilities to *"cumulatively"* store these platform metrics into the PCRs, to overcome the limitation on the total number of protected registers. Integrity Measurement is the operation used to evaluate the trust level of the platform: each component in the system start-up process measures the to-be-loaded component before transferring the control of the platform to it and these measures are stored into the PCRs. This measurement process is defined as *transitive trust*, and it forms the basis for Remote Attestation.

The protected capabilities allow free read access to the PCRs by the platform's users, but direct writing is prevented. Hence PCRs act as accumulators: when the value of a register is updated, the new value depends both on the new measure and on the old value, to guarantee that once initialized it is not possible to forge the value of a PCR. This is the *extend* operation that works as follows:

$$\text{PCRnew} = \text{SHA\_function}(\text{PCRold} \mid\mid \text{measured\_data}) \tag{1}$$

where *PCRold* is the value present in the register before the extend operation, *SHA_function* is the cryptographic hash function (i.e. SHA-1 in case of TPM version 1.2, both SHA-1 and SHA-256 in case of TPM version 2.0), || is the concatenation operator, and *measured_data* is the new measure to be inserted. Most PCRs (typically PCR-0 to PCR-15) have persistent values until the whole platform is reset (e.g. rebooted). Thus, even in case of system compromise, an attacker cannot forge PCR values at his favour once the system is rebooted. The PCR value at a given time is the result of ordered extend operations performed on the platform's measurements, and the cryptographic nature of this operation makes it impossible to retrieve any individual measure of the platform status. Because of this, integrity measurement architectures typically log each integrity measure in order to reconstruct the final PCR value starting from the individual digests. To trust the operations of protected capabilities, the TCG defines three so-called *Root of Trust* (RoT), i.e. components of a TP meant to be trusted because their misbehaviour may not be detected. The *Root of Trust for Measurements* (RTM) implements an engine capable of making inherently reliable integrity measurements and it is also the root of the chain of transitive trust. The *Root of Trust for Storage* (RTS) securely holds the integrity measurements (or a summary and sequence of those values) and protects data and cryptographic keys used by the TP that are held in external storage. The *Root of Trust for Reporting* (RTR) is capable of reliably reporting the measurements held by the RTS to external entities (e.g. the platform's users). A TPM can provide both RTS and RTR, while the RTM is typically implemented by the BIOS and it is activated as soon as the system is booted.

2.2.1. The Remote Attestation process

A RA operation is initiated by a remote party, the Verifier, to request evidence about the integrity state to an attesting party, the Attester. The most appropriate format for an integrity evidence is the TCG *Integrity Report* (IR) [23], which comprises the values stored in the PCRs and their digital signature computed with a key that never leaves the TPM (as a guarantee that the key can be used only by the TPM itself, as it is held in the device memory). However, note that the PCRs value depend not only on the software measured on the platform, but also on the order of the extend operation performed on these measurements. Hence, it is a very difficult task to know which the *"good"* and *"bad"* PCR values are. To overcome this problem, the binary attestation approach provides a run-time measurement list containing the names and digests of the measured components and their configuration files. This approach uses the TPM only as a trust anchor (to store the PCR values and to generate the digital signature over them) and permits implicit proof of the integrity of the run-time measurement list. In this case, the Verifier must be able of comparing measured software components in the run-time measurement list against a white-list of known-good values.

2.2.2. Run-time Integrity Measurement Architecture

We use the Linux IMA [24] technology for creating the measurement list because it does not require any modification to Linux (as it is already integrated into the Linux kernel since version 2.6.30) and it is considered by the TCG as a favourable technology for run-time integrity verification [25]. In fact, IMA maintains a TCG-compliant Integrity Measurement Log and it is the state of the art amongst the static measurement mechanisms. It is the first practical work to extend the TCG trust measurement concepts to dynamic executable content from the BIOS all the way up to the application layer, which makes the Remote Attestation technique much more suitable in commodity platforms. In this sense, it makes TC-compliant system practical into a real-world scenario. Once activated, IMA will start measuring the accessed files according to the criteria specified in its application policy, which is defined by the administrator. The IMA policy can be used to constrain the type of measurable events, such as memory-mapped files and executable files. Each digest is calculated before

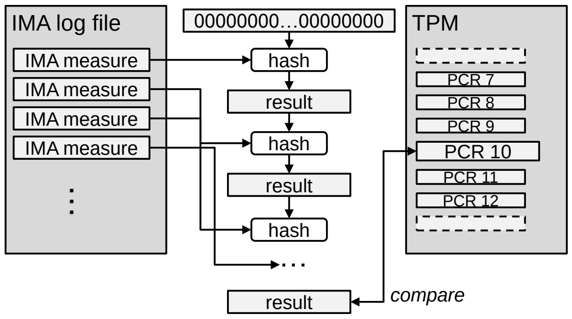


Figure 2: IMA verification process

the measured component takes control of the platform and its value extended into a PCR, hence guaranteeing that the obtained measurement cannot be tampered by the component itself. IMA maintains an aggregation of the integrity measurement over one of the available registers in the TPM, i.e. PCR-10 by default. The latter will be included in the evidence produced by the TPM, so that the trustworthiness of the measurement list can be cryptographically verified. A challenging party can attest the system run-time integrity using the IMA measurement list together with the expected PCR values. The verification process simulates the extend operation, by hashing the concatenation of each measurement and the previous hashed value. If the result matches the verified value in PCR-10, then it can be concluded that the IMA measurement list has not been tampered (Figure 2).

IMA does not easily translate to a virtualised environment by default. The integrity state information generated by the services running inside a virtualised instance cannot be directly extended into the hardware TPM, thus introducing security concerns. For example, even if IMA is activated inside the VM, the IMA measurement list cannot be explicitly or implicitly authenticated by the TPM in a scalable manner. Problems are either no direct interaction between the OS in the VM and the TPM, or the lack of enough PCRs to support the quantity of VMs typically running in a server. For these reasons, the chain of trust is broken at the virtualisation layer and the TPM can be used only to attest the integrity of the hypervisor. For example, this is the Trusted Compute Pools approach proposed by Intel [26], which offers trust in the computing nodes but not in the hosted VMs. In this work we focus on enabling IMA support for containers, which leverage the sharing of the kernel with the host differently than VMs. Because of this, we are able to keep the chain of trust intact regardless of the virtualisation layer.

3. DIVE architecture and monitoring process

We propose DIVE, a solution to provide integrity evidence for a lightweight cloud environment. The solution can be applied separately to each compute node equipped with only a single TPM, and it enables integrity verification of both the host, the container engine and the running containers. The evidence of the integrity state of the services running inside these containers is authenticated by the TPM. Thus, it can be directly used for verification in the Remote Attestation phase. A bonus property provided by DIVE is the possibility to distinguish which container is compromised. This is a significant feature because, once a container is compromised, it can immediately be stopped and readily replaced by a freshly created one, with no need to reset the whole platform. In this way, our solution greatly improves the Remote Attestation efficiency, and makes it viable for practical applications.

3.1. The DIVE architecture

Our schema contains three components: the Verifier, the Attester, and the Infrastructure Manager, as depicted in Figure 3. The Attester is the target of the Remote Attestation process, equipped with a TPM and running the integrity measurement framework, i.e. IMA. The RA agent is the component that enables the RA process on the Attester side, as it issues commands to the TPM, retrieves the list of measurements from the IMA module and is in charge of interacting with the Verifier. Moreover, the Attester runs the Docker container engine. In a cloud scenario, each Attester represents a compute node where lightweight

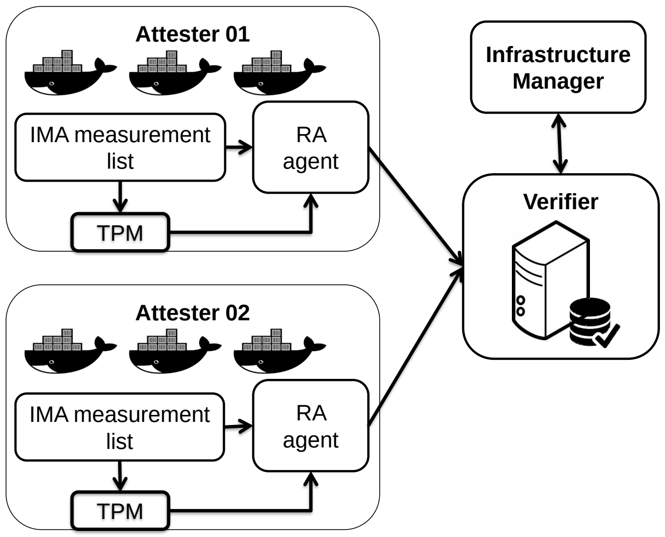


Figure 3: The DIVE architecture

virtual instances are run. The Verifier is the core of the architecture, since it is in charge of deciding the integrity state of each Attester and the containers running in it. The Infrastructure Manager creates the required containers for the users and keeps track of both the *Universal Unique Identifiers* (UUID) of the containers and the nodes hosting them.

When a user asks for the integrity of a service deployed in some containers running in this cloud, the Infrastructure Manager initiates the Remote Attestation process: it sends to the Verifier the list of containers to be checked, along with the list of the machines hosting these containers and ask for their integrity state. The Verifier, after receiving the Remote Attestation request, first contacts the Attesters in the list, asking them to send back their integrity reports that include evidence for both the host system and all the containers running in it. With the knowledge of the container list, the Verifier checks the measures belonging to the containers of interests against a white-list, e.g. a reference database of distribution packages. Finally, the Verifier generates an attestation report and sends it back to the Infrastructure Manager. Since the Infrastructure Manager oversees the management of all the containers, it can start a roll-back strategy if it notices the compromise of any container. For example, the manager can terminate the compromised container and start a new one without rebooting the whole system. However, if the host OS (i.e. the one running Docker itself) is compromised, then the whole system should be rebooted because compromised containers may not be detected any more (as the Attester's Integrity Report could have been manipulated by an attacker).

3.2. DIVE monitoring process

DIVE can be implemented in a lightweight cloud infrastructure in order to introduce a monitoring process based on run-time integrity verification of both the physical and the virtual infrastructure. The Infrastructure Manager is in charge of periodically query the Verifier to assess the trustworthiness of the platform. When the Verifier receives an integrity verification request from the Infrastructure Manager, the seven steps in Figure 4 must be performed before a verdict can be issued. To minimise performance loss and attack surface of the attesting platform, the RA agent does not accept incoming attestation requests from third parties, rather it periodically polls the Verifier (specifically registered at setup for this task) at a predefined interval. Once a RA request is present (step 1), the agent issues a quote operation to the TPM (step 2) for getting the PCR values and uses the result along with the recorded IMA measures to create the IR (step 3). The time to perform a quote operation is about 2 s because of the TPM device latency, while the time to prepare the IR is proportional to the number of IMA measures. Thus, the size of the IR is also proportional to the number of IMA measures and so is the time needed to transmit the IR from the Attester to the Verifier (step 4). When the IR is received, the Verifier checks the digital signature of the quote output against the public-key certificate of the Attester, stored when the host machine was registered. Next the consistency of the IMA measurement list is checked against the PCR values (step 5). When these steps are completed, the measurement list is distilled to retain only the measures concerning elements of interest (i.e. the host system plus all the containers present in the verification request from the Infrastructure

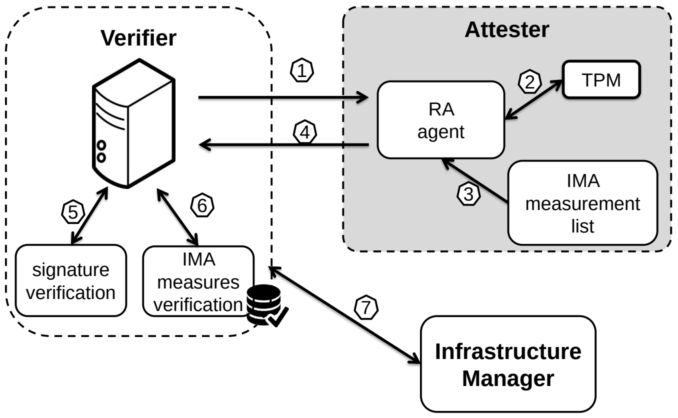


Figure 4: DIVE Remote Attestation work-flow

Manager). This reduced list of measures is passed to the IMA verification logic to verify them against the reference database (step 6). At this point, the Verifier can return the integrity verification result to the Infrastructure Manager (step 7).

It is to be noted that the DIVE work-flow requires the Attester to provide the full IMA measurement list to the Verifier at each attestation, to ensure that the containers and the host system are running white-listed software. In case a binary is either updated or patched at run-time, it would be flagged as untrusted at its next execution in case the white-list is not updated with the latest digest for such binary. Therefore, it is mandatory that the white-list is updated each time the host system is updated, or a false positive will be triggered. Nevertheless, this limitation is not considered a major drawback of our proposed solution in its target environment, i.e. a cloud lightweight compute platform, wherein updates should not be as frequent as they are in consumer machines to enhance the platform's stability. The same limitation applies to containers, although containers' images are not expected to be updated as frequently as the host system, especially in a cloud scenario (where the images of the containers are typically preloaded in the image service managed by the Infrastructure Manager).

4. Prototype of the architecture

In this section we describe the mapping of architectural components to software modules required for the DIVE functional prototype and the necessary modifications to the software to implement the proposed work-flow.

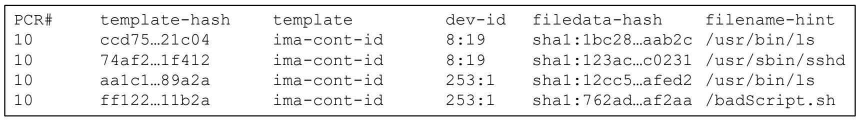
4.1. Prototype software and tools

We use the *OpenAttestation* (OAT) SDK v1.7 [27] to implement the Remote Attestation service in our infrastructure. Started by Intel, OAT provides a complete Remote Attestation implementation which is compliant with TCG specifications (e.g. the integrity report template sent from the Attester to the Verifier) until version 1.7, while versions from 2.0 are not TCG-compliant any more. OAT also includes a web interface to present the history of integrity reports and a set of RESTful APIs for easy integration in other tools. OAT provides both client and server components of the Remote Attestation process, respectively in the Attester and Verifier, which are named HostAgent and Attestation Server. OpenAttestation relies on *Transport Layer Security* (TLS) to provide encryption of communications within the Remote Attestation workflow. In fact, once the HostAgent is initialised, it is configured with the X.509 certificate of the Attestation Server so that it can encrypt the Integrity Report with the remote party public key. Moreover, OpenAttestation also supports two-way TLS for the Attestation Server Query APIs, so that only verified entities (such as the Infrastructure Manager) can be allowed to request the trust level of the infrastructure. Then, we exploit the Linux IMA module and the TPM 1.2 device for integrity verification by the Attester. Although a more recent version of the TPM exists, namely the 2.0 release, we adopt the previous iteration for several reasons. First, this is the only version supported by OAT. Moreover, the current IMA implementation only extends measurements to the SHA-1 bank (which is the only choice with TPM 1.2), hence it cannot benefit from the extended PCR banks of the 2.0 version. Finally, the latest release of the TCG integrity report specifications at the time of writing [23] only supports the TPM 1.1 and 1.2 versions.

Finally, the Infrastructure Manager can be mapped on a container management engine or orchestrator platform such as OpenStack, which encompasses several projects that tackle support for Docker containers in an orchestrated cloud environment [28], or the Kubernetes container management engine [29]. Our initial prototype focuses on the implementation of the Attester and Verifier elements of the DIVE architecture, hence it does not take into account integration with any Infrastructure Manager, which is left for future work.

4.2. Attester prototype

The Attester is a physical host machine wherein the Docker Container Engine is is deployed. IMA is the core tool used by the Attester for logging the integrity information (loaded binaries and configuration files) of the containers and the host system. Given the internal working of Docker, the containers invoke directly the functions in the underlying kernel of the host system. Thus, when binaries and files in the containers are loaded into memory, these operations will be automatically captured by the IMA module present in the host system: the corresponding digests are appended to the IMA measurement list and extended into a PCR (by default, PCR-10) to ensure the integrity of the IMA measurement list. This standard feature is the basis of our solution and, to adapt its usage for a container-enabled platform, we created a new IMA template, i.e. a custom format for each entry in the IMA measurement log. Our template adds a new attribute, called `dev-id`, which correlates each binary with its execution virtual device identifier, thus allowing to identify the corresponding container. This feature exploits the Device Mapper storage driver for Docker, as mentioned in Section 2.1.2, which assigns a virtual device identifier to all the processes running in each container. Figure 5 displays an example of IMA measurement log with the custom template and several measures: two of them belong to the host system, which has the device ID `8:19`, while the other three belong to two containers, specifically those with device ID equal to `253:1` and `253:2` (fourth column). When the whole measurement list is sent to the Verifier, it will decide if the container with device ID `253:1` needs to



```
PCR#   template-hash   template       dev-id  filedata-hash     filename-hint
10     ccd75…21c04     ima-cont-id    8:19    sha1:1bc28…aab2c  /usr/bin/ls
10     74af2…1f412     ima-cont-id    8:19    sha1:123ac…c0231  /usr/sbin/sshd
10     aa1c1…89a2a     ima-cont-id    253:1   sha1:12cc5…afed2  /usr/bin/ls
10     ff122…11b2a     ima-cont-id    253:1   sha1:762ad…af2aa  /badScript.sh
```

Figure 5: IMA measurement log with custom template for container identification

be checked. In this way, not only the Verifier knows which binaries have been executed, but it can also identify the container or host system where they are executed. This feature brings a key advantage: if a container loses its trusted state at any point, then the Infrastructure Manager does not need to reset the whole Attester platform to restore trust. On the contrary, it just needs to destroy the untrusted container and start a new one. Hence, this feature makes the Remote Attestation technique much more appealing from a performance point of view and applicable in a real case scenario. The default IMA process measures every file that respects the IMA policy only the first time that it is loaded into memory, via its *inode*. Hence, the inode of the file must vary if the file has been modified. IMA internally implements a global measurement cache, i.e. an hash-table that keeps record of software events together with their template entry. To be included in the measurement log, a file must conform to the IMA policy, hence its digest must not be included in the hash-table, and its inode must not have been already measured by IMA.

Additionally, we also modified the OAT HostAgent application (which is executed in the Attester) so that it can provide the mapping between container UUIDs and device IDs in the XML Integrity Report, as shown in Figure 6. The new elements are named `<Container>`, providing the mapping between a Docker container UUID and the associated virtual device ID created by Device Mapper in the host system, and `<Host>`, containing the list of all physical devices ID associated to the host system.

Finally, we extended the Docker *Command Line Interface* (CLI) to retrieve the mapping between each container UUID (provided to the Verifier by the Infrastructure Manager) and its device ID in a timely manner.

```
<Container Id ="8948d6f37d41">
   <DevId>253:1</DevId>
</Container>
<Container Id ="1beb7b9c05s6">
   <DevId>253:2</DevId>
</Container>
<Container Id ="2f9695f9db36">
   <DevId>253:3</DevId>
</Container>
<Host>
   <DevId>8:0</DevId>
   <DevId>8:1</DevId>
</Host>
```

Figure 6: Excerpt from the modified OAT Integrity Report

4.3. Verifier prototype

OAT is the core tool for providing the Remote Attestation function on the Verifier side. The Verifier only knows the list of the container UUIDs, which is sent from the Infrastructure Manager. We extended the OAT Attestation Server to map container UUIDs to the device IDs retrieved in the OAT HostAgent. Once the Verifier retrieves the list of measurements of the containers by the Attester, it has to match them against a reference white-list database.

In a previous work [30], an implementation of the binary attestation feature with the help of IMA has been developed. This implementation creates a reference database storing the names and digests of all the known *"good"* binaries. For example, we could initially populate the database with all the elements of available packages in the official repositories for Linux distributions. One drawback of this solution is that it considers the platform as a whole: if the integrity report contains just one unknown digest for a loaded binary or applied configuration, the Verifier will consider the whole platform as untrusted and a trusted state can be restored only by resetting the whole platform. Unfortunately, resetting a physical platform hosting tens or hundreds of VMs or containers is not acceptable in a real-world scenario when the problem is affecting just one virtual instance. To solve this problem, we extended the original implementation with a new analysis type at the Verifier, which requires a list of the containers as parameter to be checked with the nodes hosting them. Such list is created by a manager based on the target virtual instances to be verified for a specific service or user. The new analysis type reduces the full IMA measurement lists by keeping, for each node, the measures for the containers of interest plus those of the host system (since it is the base for running the containers). Afterwards, the Verifier compares the list of measures with the known good values in the reference database. Since each measure in the extended IMA template references a specific container, the result of the verification is related only to attested containers. For example, if only one container is compromised and it is present in the list of target containers, then the attestation result will be untrusted. On the contrary, if the compromised container is not being attested (e.g. the manager does not care about its state because it is not involved in any sensitive operation), the overall system status will remain trusted. In the case of untrusted result, the Verifier reports which measures are unknown from the reference database and which container they belong to, so the manager can proceed to destroy only the corrupted container in the attested host, with no need of rebooting the whole platform.

5. Performance evaluation

We have performed a preliminary evaluation of the DIVE technology, mainly focused on its performance impact on a single host deployment of Docker containers. Our goal is to demonstrate the negligible impact of this solution compared to an insecure Docker deployment.

Our first test focuses on the performance of DIVE when applied to a variable number of containers at different moments of their life-cycle, i.e. when started, stopped, or removed by the Docker engine. Next, we evaluate the performance impact of DIVE on both the Verifier and the Attester separately, with a varying number of attested containers. In our evaluation, we limit the maximum number of active containers to 512, as higher values have shown a degradation in system performance and allocation of resources (e.g. virtual network interfaces)

To evaluate the performance of our solution, we set up the following testbed: an Attester, running on a node with an Intel Core i7-4600U CPU (2 core, 4 threads) and 16 GiB of RAM, and a Verifier running in a VM with 2 Intel Xeon CPU @2.4 GHz and 4 GiB of RAM. Both machines are running CentOS 7, with a custom kernel based on version 4.4 that includes the IMA modifications. The Attester is running Docker Consumer Edition version 18.03.1-ce, with Device Mapper library version 1.02.110, driver 4.34.0.

The activation of IMA affects the performance of Docker because IMA measures all loaded executables and their configurations files, and then extend these measures into the TPM. Hence to provide adequate coverage but to avoid measuring unnecessary files, we configured IMA prescribing to measure only files mapped in memory as executable, files executed by the `execve()` system call in the kernel and files labelled as `CONFIG` (i.e. configurations).

In our first test, the number of IMA measures grows linearly with the number of containers launched in the platform. Of course, if more complex services are deployed inside the containers, the number of measured binaries and static configuration files would increase even by orders of magnitude. For testing the performance penalty introduced by IMA and RA in the attesting platform, we considered a deployment scenario comprising 512 containers. We repeated each test ten times, to get statistically meaningful results, and executed the test set (comprising ten runs) first without IMA and RA, then with only IMA active, and finally with both IMA and RA active. Three essential operations are involved in the test: run (to start a container), stop (to send `SIGTERM` to the process running inside a container) and remove (to remove a container along with its assigned resources, e.g. disk storage and network bridge). The average time for these operations is depicted in Figures 7, 8, and 9, where the X-axis shows the number of active containers and the Y-axis is the time to finish the operation. These graphs are to be interpreted as time needed to perform an operation given a certain number of containers already active. For example, values for 256 containers represent respectively the time needed to create a new container (hence to have 257 active containers), to stop a container (hence to have 256 containers with only 255 active), and to remove a container (hence to have 255

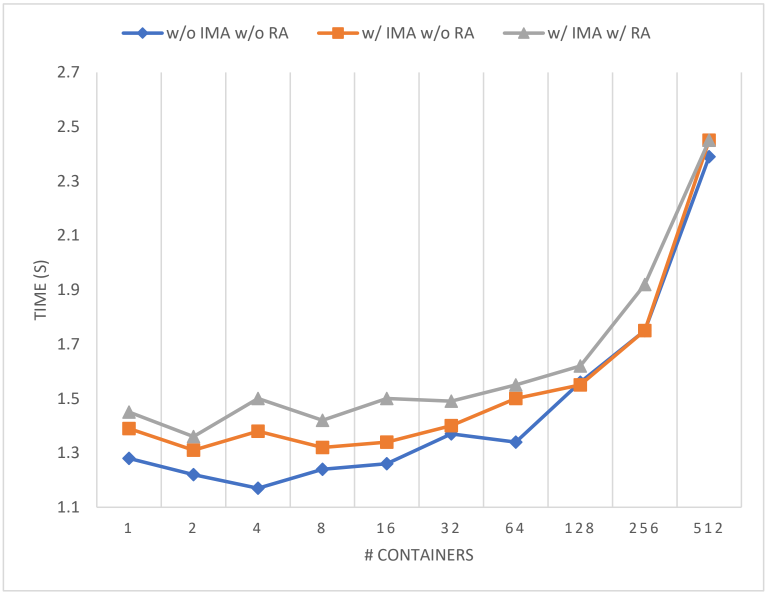


Figure 7: Time to start containers with and without IMA and RA (in seconds)

active containers). The main difference in performance with and without IMA is related to the start of a new container (Figure 7) which requires about 80 ms more with IMA and 170 ms in the case both IMA and RA are activated. Apart for statistical fluctuations, the overhead introduced by IMA when starting a new container is independent of the number of active containers. This is reasonable because the start operation directly adds workload to the attesting platform and each start operation needs to measure and extend new IMA measures into the TPM. In our test, we can estimate that each new measure requires about 11 ms. An interesting observation is that the start time of the first four containers without IMA is decreasing because the CPU of the attesting machine has two cores and supports four threads, hence the first four containers have the privilege to share the resources equally without competing for CPU cycles. The behaviour changes and is more unpredictable as another important process (IMA itself) comes into play. The differences for the stop and remove operations with and without IMA and RA are minor (if any), since these two operations do not

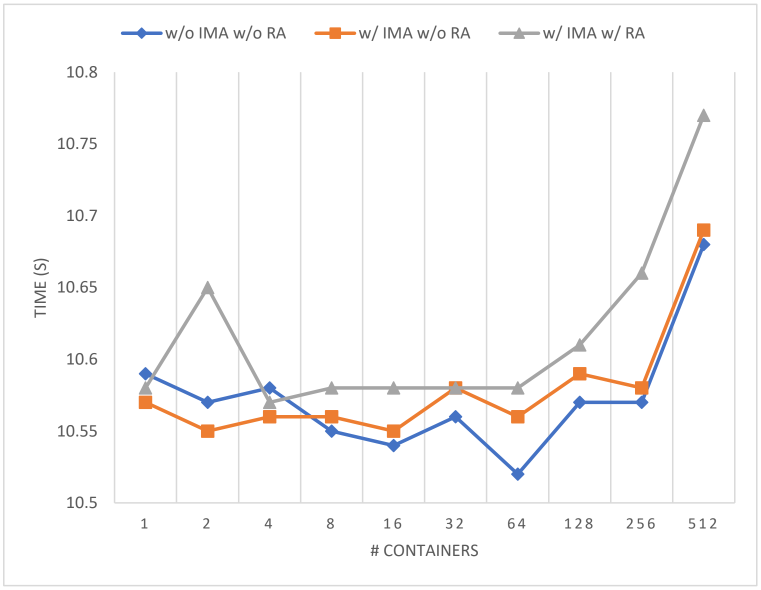


Figure 8: Time to stop containers with and without IMA and RA (in seconds)

involve any IMA or RA functionality. The time to stop a container is always around 10 s (Figure 8) because the Docker stop operation sends `SIGTERM` to the main process of the container. However, in our test, this signal cannot stop the process executed in the container, so it needs to wait the predefined threshold (with default value 10 s) to send `SIGKILL` and kill the whole container process. The remove operation removes the resources assigned to a container and the time needed grows linearly with the number of active containers (Figure 9) and doesn't show any significant difference among the three different set-ups.

Table 1 presents the results of our test about the global performance of the integrity verification process with varying number of active containers (i.e. variable number of IMA measures, since the relation is linear). The time is given both as a total and as two components related to the Attester and the Verifier, since the time for transmitting the IR is very small and hence negligible: with 512 containers the number of IMA measures in our test is around 4500 and the size of the IR is about 1.38 MiB, which only needs around 0.1 s to be transmitted.

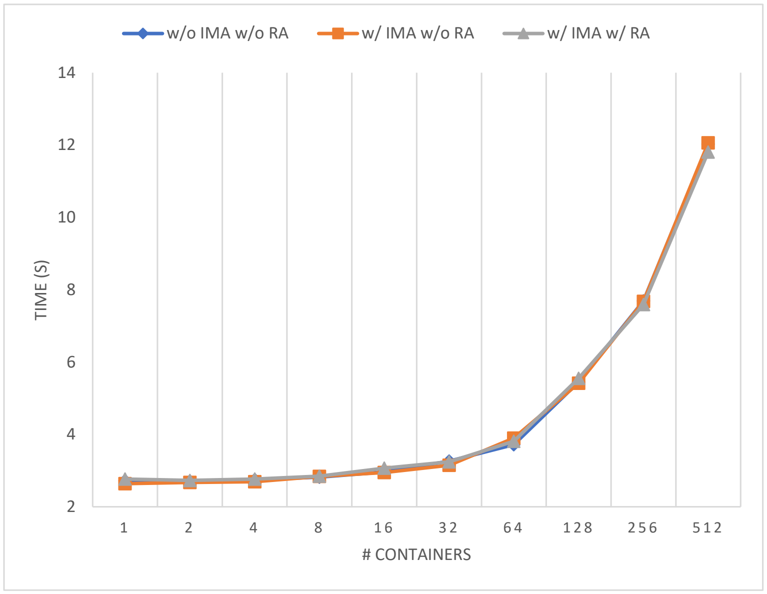


Figure 9: Time to remove containers with and without IMA and RA (in seconds)

	# containers	**0**	**32**	**128**	**256**	**512**
	Attester	3.71	3.80	4.05	4.99	6.56
time	**Verifier**	1.56	1.63	1.77	1.95	2.35
	total	5.27	5.43	5.82	6.94	8.91

Table 1: Average time (in seconds) required to complete an integrity verification request.

The total time (and the individual components) grows linearly with the number of active containers (given that in this case each container produces the same number of IMA measures). The time taken by the Attester is related to the quote operation and the generation of the IR (i.e. steps 2 and 3 in Figure 4), while the time taken by the Verifier includes the effort to verify the signature of the IR and to compare the IMA measures with the reference database (i.e. steps 5 and 6 in Figure 4). The other steps (i.e. steps 1 and 4 in Figure 4 have small influence on the overall time. The time to forward the attestation result to the Infrastructure Manager (i.e. step 7 in Figure 4) is not considered in the overall evaluation of our solution, as it is not part of the Remote Attestation workflow. Even with 512 active containers, the time needed to attest them is less than 10 s, which we deem a good result considering the integrity guarantee provided by our solution. In case this performance is not considered adequate, there are margins for improvement. On the Attester side, the hard limit is the time taken to perform the digital signature over the IR by the TPM, which in our machine takes about 2 s. The rest of the time is spent in creating the IR and this time can be reduced by creating differential reports (i.e. containing only those measures added since the last IR), which would benefit also the Verifier as it would have to perform less queries to the white-list database. In case a high-frequency verification is desirable, then dedicating a whole processor core to the Docker host system would be an option, so that the OAT agent would not have to compete with the containers for the CPU. Additionally, re-implementing the OAT agent as native code would improve performance as current implementation is in Java, which incurs overhead in speed and memory size.

Finally, we evaluate the resource consumption at the Verifier side in terms of percentage of CPU usage and RAM utilisation at the increase of the number of containers. In this regard, we measure the Java process of the OAT Attestation Server during the whole Remote Attestation workflow in ten separate runs, and we calculate the average value for each measure. It is to be noted that the Java process utilises a single core of the CPU on the Verifier VM. The result of this test is displayed in Table 2, which shows that container integrity verification significantly impacts the CPU utilisation at the Verifier side, if compared to host-only attestation. This is due to the fact that the Verifier needs to validate the signature of a longer Integrity Report, and to compute the aggregate value over a larger list of IMA measurements in order to attest the integrity of all the containers. On the other hand, the memory consumption at the increase of running containers is not affected as much, as the Verifier does not keep in memory the list of measurements at each attestation process.

# **containers**	**0**	**32**	**128**	**256**	**512**
% **CPU** (1 core @ 2.4 GHz)	3.51	13.01	14.55	18.10	26.33
% **RAM** (4 GiB)	7.74	8.09	9.11	12.01	15.10

Table 2: Average resource consumption of the Verifier at the increase of running containers.

6. Related work

TCG-based integrity verification for Linux containers has been discussed in scientific literature. Sun et al. [31] have proposed to include a Linux namespace to isolate the security context of each container. In their proposal, the authors suggest that such namespace could be exploited to keep independent security policies for each container, allowing each of them to independently govern their security. The authors propose to isolate the IMA measurements and *Mandatory Access Control* (MAC) rules enforced by the *Linux Security Modules* (LSM) framework, and develop a prototype for both IMA and the AppArmor LSM[32]. Finally, the authors introduce a policy engine in their design to refine security policies on shared objects (e.g. files) defined in separate security contexts. Similarly to DIVE, the authors rely on the global IMA measurement cache, although the visibility of IMA log entries is specific to each security namespace. This means that the host itself does not have visibility over the measurements executed in each container. In DIVE, the Attester must be able to retrieve the measurements executed in each container to perform the Remote Attestation process.

Arnautov et al. [33] have proposed the SCONE technology to secure Linux Containers leveraging the Intel *Software Guard Extensions* (SGX) technology. SGX is a set of instructions for the Intel CPUs that allows defining secure memory regions, named enclaves, where data and code are shielded from external access. SCONE technology focuses on Docker containers that implement network services (as in our proposal), as they require a limited interface for system support. This is due to the performance overhead for context switching introduced by SGX, which typically does not execute system calls inside the enclave. Differently from our proposed solution, SCONE focuses on ensuring both integrity and confidentiality of containers' images, but it may have a more significant impact on the performance of containers. Moreover, the integrity guaranteed by SGX is limited to code and data that resides in the enclave (each enclave page size is between 64 MiB and 128 MiB and swapping pages between the secure environment and the untrusted DRAM introduces an overhead), while our solution targets integrity verification of a complete execution environment, comprising the host and several containers.

There are past works about attestation of virtual machines and we discuss here various techniques and their possible application to Docker containers. The most straightforward approach to extend the chain of trust from the TPM to the VM is to simulate a physical TPM by a *virtual TPM* (vTPM): each VM has access to its own private TPM simulated via software [34]. The overall vTPM facility is composed of a vTPM manager and several vTPM instances. Each VM has its own vTPM instance and the vTPM manager oversees creating instances and multiplexing requests from the VMs to their associated vTPM instances. The extend operation works on values stored inside the state of the vTPM instance. The guest OS needs to be equipped with a client-side TPM driver, to which the guest OS sends its TPM commands. A server-side TPM driver is running in a privileged VM (having direct access to the hardware, such as Dom0 in Xen) along with the vTPM manager. This server-side driver collects the requests from the client-side driver and sends them to the vTPM manager. The hardware TPM records the measurement of the boot process of the attesting platform and it is also used for attesting the privileged VM hosting the vTPM server-side driver and manager. Currently, this solution is popular with Xen community, as the vTPM implementation is present in the official version of this hypervisor.

Using the vTPM with Docker containers would create unnecessary overhead (due to the vTPM client-server schema and its multiplexing operations) and would require modification of the containers' images (as each image needs its own vTPM client-side driver). The approach of DIVE is more direct, does not require modification of the images, and has excellent performance. The scalability of the vTPM solution has been improved by using an event-based monitoring and push model [35], because the traditional periodic polling model does not work well with the vTPM (each VM adds effort to the attestation cost as it requires its own polling system and TPM-based signature of its IR). In this approach, a special vTPM instance supervises all the others and notifies subscribed users of any change, thus offering event-based attestation. This event-based approach could also be taken by DIVE, as it alleviates the Time-of-Measure Time-of-Report problem (i.e. delay in reporting a compromise until the next polling cycle). However, we have seen that the bottleneck at the Attester is the signature generation speed of the TPM and hence this delay problem cannot be completely avoided. Moreover, creating an IR and generating a signature over it every time an extend operation occurs could become a major burden for the Attester when hosting many containers. Since the hypervisor mediates all the VM operations, it can be modified to monitor them and offer an event-based attestation. This idea is used by an *Integrity Verification Proxy* (IVP) embedded in the hypervisor [36]: the IVP monitors the VMs integrity, while the integrity of the IVP and the host is verified by traditional attestation methods. However, monitoring the internals of VMs requires fine-grained techniques, such as Virtual Machine Introspection, that may imply extensive modifications to the hypervisor or payment of a big penalty in performance. For example, the proposers of the IVP used the *GNU Project Debugger* (GDB) to monitor the user-space VMs, pausing them until integrity verification is completed; of course, in a production environment this is unacceptable. Although the IVP is an interesting concept, for Docker integrity verification we don't need it as we have shown that all operations performed by the containers are directly available for monitoring from the host. Hence DIVE can be considered as a simple and fast implementation of the IVP idea for Docker.

Terra [37] is a trusted virtual machine monitor that isolates and protects independent VMs. However, its integrity guarantee is limited to the hypervisor and to computing and certifying the hashes of the images loaded in the VMs, but there is no attestation of the internal behaviour of the VMs. As such, it's more similar to the load-time image integrity attestation offered by Trusted Compute Pools for OpenStack [26], and by Docker Content Trust. DIVE can use the latter for image integrity verification at load time but additionally it offers continuous monitoring and reporting of the operations performed by each container. In this respect, it is better than Terra, although the latter offers other interesting security properties not relevant for the current discussion. Compared to [38], which generates a model for Docker container and verifies the deployability of the container architecture at design time, our work is more focused on verifying the integrity of the services running inside the containers in a reliable manner. Of course, these two approaches could be coupled to provide a holistic Docker container execution environment.

The Intel Open CIT framework [39] implements image integrity verification for both VMs and Docker containers, and it adopts Intel *Trusted Execution Technology* (TXT) [40] for measuring the boot state of the machine, levering the TPM for secure storage. Image integrity verification is achieved by mounting the virtual image file-system, and then statically measure the files and store their digests and paths in a Trust Policy. This is later used as a reference database to verify the image at next boot. Open CIT allows the user to choose whether to prevent any untrusted image from being powered on, or not. Compared to our solution, Open CIT extends attestation to VM images and it allows a fine tuning of the folders and files of the image file-system. Nonetheless, the Trust Policy addresses static verification of the container file-system at load-time only, while our solution can be used to periodically attest the running instances during execution. Our extended IMA verification relies on an external reference database populated with packages from software repositories, allowing attestation of instances wherein software is dynamically installed and/or updated. In Open CIT, the white-list is statically built at first load of the image and must be manually updated if any measured file is changed.

7. Conclusion and future directions

We have presented DIVE, an architecture to support integrity verification of Docker containers using well-known trusted computing techniques. With this solution, the services in the containers can be attested as if they would be running in a physical platform, and their integrity can be well understood by a third-party in a reliable manner. The capability to directly interact with the TPM makes the integrity reports from the Attesters non-forgeable, which provides strong protection towards remote attacks. DIVE is also a practical tool as it has a nearly negligible performance impact on the hosted services. This is due both to its design and the usage of mainstream tools, such as IMA (a standard feature of the Linux kernel) and OpenAttestation (a well-known tool for attestation of cloud services). The most data-intensive part of the work is offloaded to the Verifier, which is a third party not directly involved in the actual service provision. Another important feature of DIVE is that the Verifier can identify which container or hosting system is compromised. Thus, the Infrastructure Manager can take an informed decision about the roll-back strategy: disable just the compromised container and replace it with a new instance or stop all containers and restart the whole physical machine. DIVE is transparent to the services running in the containers (that don't need to be modified in any way) and interacts with a normal Docker environment. All modifications to enable integrity verification are minor and performed directly in the host system. This makes DIVE very easy to be adopted and with no impact on the hosted services. DIVE has some limitations that we plan to address in future work. First, its dependency on OpenAttestation makes it non-portable on hosts equipped with TPM 2.0. In this regard, integration of DIVE in the Open CIT load-time integrity verification process will be investigated. Moreover, our solution introduces a lock-in on the Device Mapper storage driver for Docker, although it is generally supported on Linux distributions. Finally, DIVE cannot ensure protection against in-memory attacks, which would require the introduction of separate user-space and kernel-space protections (e.g. address space randomisation). Planned future work for DIVE is twofold: performance and proactivity. Application to softwarised network environments typically requires a small footprint and fast reaction time, which can be obtained by rewriting the attestation agent and by looking to hardware trust anchors with better performance than the standard TPM chip. Being informed that a service has been compromised is important for its management but avoiding the compromise would be even better. Along this line, we want to make DIVE a proactive service by coupling it with policy enforcement solutions, such as SELinux.

Acknowledgement

The authors wish to thank Dr. Tao Su for his valuable contribution in the creation of a first prototype of DIVE.

The work described in this article has received funding by the European Union Horizon 2020 research and innovation programme, under project SHIELD (Grant Agreement no. 700199).

References

[1] Trusted Computing Group website, `https://www.trustedcomputinggroup.org`, Accessed on 21.10.2018.

[2] Trusted Computing Group, Virtualized Trusted Platform Architecture Specification Version 1.0 Revision 0.26, Tech. rep., Accessed on 30.10.2018 (2011).
URL `https://trustedcomputinggroup.org/wp-content/uploads/TCG_VPWG_Architecture_V1-0_R0-26_FINAL.pdf`

[3] Datadog, 8 surprising facts about real Docker container adoption, `https://www.datadoghq.com/docker-adoption/`, Accessed on 21.10.2018 (2017).

[4] Business Cloud News, Deutsche Telekom experimenting with NFV in Docker, `http://www.businesscloudnews.com/2015/02/09/`

deutsche-telekom-experimenting-with-nfv-in-docker/, Accessed on 21.10.2018 (2015).

[5] AT&T, Building an Enterprise-Focused Cloud Environment, http://about.att.com/innovationblog/enterprise_cloud, Accessed on 27.10.2018 (2017).

[6] M. Souppaya, J. Morello and K. Scarfone, NIST Special Publication 800-190 - Application Container Security Guide, Tech. rep., NIST, Accessed on 15.10.2018 (Sep. 2017). doi:10.6028/NIST.SP.800-190.

[7] Docker project website, https://www.docker.com/, Accessed on 15.10.2018 (2018).

[8] Address Space Layout Randomisation, https://pax.grsecurity.net/docs/aslr.txt, Accessed on 28.10.2018 (2001).

[9] S. Soltesz, M. E. Fiuczynski, L. Peterson, M. Mccabe, J. Matthews, Virtual doppelgänger: On the performance, isolation, and scalability of para- and paene- virtualized systems, http://people.clarkson.edu/~jnm/publications/paenevirtualization.pdf, Accessed on 22.10.2018 (2006).

[10] A. Hussain, Performance of Docker vs VMs, http://www.slideshare.net/Flux7Labs/performance-of-docker-vs-vms, Accessed on 22.10.2018 (2014).

[11] Linux Programmer's Manual - Cgroups, http://man7.org/linux/man-pages/man7/cgroups.7.html, Accessed on 15.10.2018 (2018).

[12] Linux Programmer's Manual - Namespaces, http://man7.org/linux/man-pages/man7/namespaces.7.html, Accessed on 15.10.2018 (2018).

[13] Linux Programmer's Manual - Capabilities, http://man7.org/linux/man-pages/man7/capabilities.7.html, Accessed on 15.10.2018 (2018).

[14] Rkt project website, https://coreos.com/rkt/, Accessed on 15.10.2018 (2018).

[15] Linux Containers project website, https://linuxcontainers.org/, Accessed on 15.10.2018 (2018).

[16] LXD project website, https://linuxcontainers.org/lxd/, Accessed on 15.10.2018 (2018).

[17] Unikernels website, http://unikernel.org/, Accessed on 29.10.2018.

[18] Docker storage drivers, https://docs.docker.com/storage/storagedriver/select-storage-driver/, Accessed on 21.10.2018.

[19] Docker Content Trust, https://docs.docker.com/engine/security/trust/content_trust/, Accessed on 15.10.2018 (2018).

[20] M. Diego, Introducing Docker Content Trust, https://blog.docker.com/2015/08/content-trust-docker-1-8/, Accessed on 27.10.2018 (2015).

[21] R. Yeluri and A. Gupta, Trusted Docker Containers and Trusted VMs in OpenStack, https://01.org/sites/default/files/openstacksummit_vancouver_trusteddockercontainers.pdf, Accessed on 27.10.2018 (2015).

[22] Trusted Computing Group, TCG Architecture Overview, Version 1.4, Tech. rep., Accessed on 30.10.2018 (2007).
URL https://trustedcomputinggroup.org/wp-content/uploads/TCG_1_4_Architecture_Overview.pdf

[23] Trusted Computing Group, Integrity Report Schema, Specification Version 2.0, Revision 5, Tech. rep., Accessed on 30.10.2018 (2011).
URL https://trustedcomputinggroup.org/wp-content/uploads/IWG_Integrity_Report_Schema_v2.0.r5.pdf

[24] R. Sailer, X. Zhang, T. Jaeger, L. van Doorn, Design and implementation of a tcg-based integrity measurement architecture, in: 13th USENIX

Security Symposium, USENIX Association, San Diego (CA, USA), 2004.
URL https://www.usenix.org/legacy/publications/library/proceedings/sec04/tech/full_papers/sailer/sailer.pdf

[25] Trusted Computing Group, TCG Guidance for Securing Network Equipment Using TCG Technology Version 1.0 Revision 29, Tech. rep., Accessed on 30.10.2018 (2018).
URL https://trustedcomputinggroup.org/wp-content/uploads/TCG_Guidance_for_Securing_NetEq_1_0r29.pdf

[26] Intel, Creating trust in the cloud - White Paper, http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/creating-trust-in-cloud-ubuntu-intel-white-paper.pdf, Accessed on 27.10.2018 (2013).

[27] OpenAttestation project website, https://github.com/OpenAttestation/, Accessed on 21.10.2018.

[28] Exploring Opportunities: Containers and OpenStack, https://www.openstack.org/assets/pdf-downloads/Containers-and-OpenStack.pdf, Accessed on 21.10.2018.

[29] Kubernetes project website, https://kubernetes.io/, Accessed on 29.10.2018.

[30] E. Cesena, G. Ramunno, R. Sassu, D. Vernizzi, A. Lioy, On scalability of remote attestation, in: STC'11 - 6th ACM Workshop on Scalable Trusted Computing, ACM, Chicago (IL, USA), 2011, pp. 25–30. doi:10.1145/2046582.2046588.

[31] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu, T. Jaeger, Security namespace: Making linux security frameworks available to containers, in: 27th USENIX Security Symposium, USENIX Association, Baltimore (MD, USA), 2018, pp. 1423–1439.

URL https://www.usenix.org/conference/usenixsecurity18/presentation/sun

[32] AppArmor, https://wiki.ubuntu.com/AppArmor, Accessed on 15.10.2018 (2018).

[33] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, C. Fetzer, SCONE: Secure linux containers with intel SGX, in: OSDI'16 - 12th USENIX Symposium on Operating Systems Design and Implementation, USENIX Association, Savannah (GA, USA), 2016, pp. 689–703.
URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[34] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, L. van Doorn, vTPM: Virtualizing the Trusted Platform Module, in: 15th USENIX Security Symposium, USENIX Association, Vancouver (B.C., Canada), 2006, pp. 305–320.
URL https://www.usenix.org/legacy/events/sec06/tech/full_papers/berger/berger.pdf

[35] K. Goldman, R. Sailer, D. Pendarakis, D. Srinivasan, Scalable integrity monitoring in virtualized environments, in: STC'10 - 5th ACM Workshop on Scalable Trusted Computing, ACM, Chicago (IL, USA), 2010, pp. 73–78. doi:10.1145/1867635.1867647.

[36] J. Schiffman, H. Vijayakumar, T. Jaeger, Verifying system integrity by proxy, in: TRUST'12 - 5th International Conference on Trust and Trustworthy Computing, Springer, Vienna (Austria), 2012, pp. 179–200. doi:10.1007/978-3-642-30921-2_11.

[37] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, D. Boneh, Terra: A virtual machine-based platform for trusted computing, in: SOSP'03 - 19th ACM

Symposium on Operating Systems Principles, ACM, Bolton Landing (NY, USA), 2003, pp. 193–206. doi:10.1145/945445.945464.

[38] F. Paraiso, S. Challita, Y. Al-Dhuraibi, P. Merle, Model-driven management of docker containers, in: 9th IEEE International Conference on Cloud Computing, San Francisco (CA, USA), 2016, pp. 718–725. doi:10.1109/CLOUD.2016.0100.

[39] Open CIT 3.2.1 Product Guide – Image Integrity (VM and Docker), https://github.com/opencit/opencit/wiki/Open-CIT-3.2.1-Product-Guide#60-image-integrity-vm-and-docker, Accessed on 19.10.2018.

[40] Intel Trusted Execution Technology (Intel TXT) Enabling Guide, https://software.intel.com/en-us/articles/intel-trusted-execution-technology-intel-txt-enabling-guide, Accessed on 30.10.2018.