Container-IMA: A privacy-preserving Integrity Measurement Architecture for Containers

Wu Luo, Qingni Shen†, Yutang Xia, Zhonghai Wu†
Peking University, China
{lwyeluo,qingnishen,ytxia,wuzh}@pku.edu.cn
†*Corresponding author*

Abstract

Container-based virtualization has been widely utilized and brought unprecedented influence on traditional IT architecture. How to build trust for containers has become an important security issue as well. Despite the fact that substantial efforts have been made to solve this issue, there are still some challenges to be handled, i.e. how to prevent from exposing information of the underlying host and other users' containers to a remote *verifier*, how to measure the integrity status of a designated container along with its reliant services in the underlying host and generate a hardware-based integrity evidence. None of the current solutions can counter these challenges and guarantee efficiency simultaneously.

In this paper, we present Container-IMA, a novel solution to cope with these challenges. We firstly analyze the essential evidence to validate the integrity of a designated container. Afterwards we make a division of the traditional Measurement Log (ML), which ensures privacy and decreases the latency of attestation. A container-based Platform Configuration Register (cPCR) mechanism is introduced to protect each ML partition with a hardware-based Root of Trust. The attestation mechanism is proposed as well. We implement a prototype based on Docker. The experiment results demonstrate the effectiveness and efficiency of our solution.

1 Introduction

Container-based virtualization technologies, e.g. Docker [18], LXC [25] and rkt [39], have become more and more prevalent, as they offer a light-weight virtualization approach to run multiple environments using the host kernel [20, 38, 42, 53]. However, since all containers share the same operating system (OS) kernel, the rapidly developing container technologies have introduced many security issues [11, 43], such as insecure production system configuration, vulnerabilities inside the images, and vulnerabilities directly linked to Docker, etc. Building trust for containers is a promising countermeasure to enhance security, as it can notify a remote verifier whether a container has genuinely enforced proclaimed security-enhancement components and other unnecessary or adverse components have not loaded.

The major existing mechanisms [9, 15, 22, 26, 50] to build trust are based on Trusted Computing technology [37]. Trusted Computing technology provides a hardware-based solution to validate the integrity of physical platforms. The *Chain of Trust (CoT)* is built from the *Root of Trust (RoT)*, which contains a chip called Trusted Platform Module (TPM) [5], embedded in the target platform (*prover*) and is trusted by default. When the *prover* powers on, all components in the boot time will be extended to *CoT*, including BIOS, GRUB and OS kernel. With the help of Integrity Measurement Architecture (IMA) [46], the *CoT* finally reaches the application layer and measures all software components loaded in *prover*. Any measured component is not only recorded into the Measurement Log (ML), but also aggregated into Platform Configuration Registers (PCRs) inside a TPM. Furthermore, the remote user (*verifier*) can perform Remote Attestation to collect ML and validate it against PCRs. If it matches, the *verifier* compares each entry with his expectation to determine the *prover*'s integrity.

Recently, researchers have made substantial efforts to build trust for containers. The mainstream mechanisms among them [7, 9, 15, 22, 26, 50] are based on the following projects or technologies: vTPM [10], IMA, rkt and SGX [6]. Trusted Computing is the bedrock for the first three projects, while SGX is designed to provide isolated areas (i.e. enclave) for applications. However, none of them is sufficient to deal with the challenges that container-based virtualization faces.

Firstly, it is very important to ensure the privacy of underlying host and other users' containers. One overarching theme of building trust for containers is to utilize IMA, since it measures the integrity of a designated platform. For example, both Tao et al. [50] and Benedictis et al. [15] measure containers' integrity through adding an additional item of IMA to indicate which container the process belongs to. Since IMA measures all components into a particular PCR and records them into a single ML, each *verifier* has to collect the entire ML when validating the integrity status of containers. Hence, every *veri-*

fier can get all information of *prover*. Adversary benefits from it to explore the vulnerabilities of *prover*, and is capable of stealing the information of other users' containers.

Secondly, in order to build trust for containers, it is essential to measure the integrity status of a designated container and its dependencies in the underlying host, as well as generating a tamper-proof integrity evidence to a remote *verifier*. However, the existing mechanisms based on *rkt* are designed to merely measure containers' boot time, such as containers' images and configurations [22]. After a container is launched, the integrity of its loaded components is ignored. On the other hand, as vTPM is useful to build separated *RoT*s for each virtual machines (VMs), some mechanisms [9, 26] are derived from vTPM. However, these mechanisms introduce additional components in the user space, e.g. the tpm emulator in [9], to manage vTPMs and bind all vTPMs to the hardware TPM. These components locate in the user space, and they are more likely to be attacked. Once these components are affected by attacker, the integrity evidence is not protected by a hardware-based *RoT*. Several other security and efficiency issues towards vTPM are discussed in [50].

Finally, ensuring efficiency is another issue. Besides the performance drop caused by vTPM, mechanisms using SGX also suffer from efficiency problems. For example, SCONE [7] is proposed to secure containers based on SGX, yet it incurs a lot of overhead with regards to the service running in native container environment [50].

To overcome the aforementioned challenges, we present a novel solution named Container-IMA. Our architecture is based on IMA, as it does not require to modify the existing architecture of containers, e.g. introducing an additional layer in the user space like vTPM. In order to preserve privacy, we firstly analyze the essential evidence to validate the integrity of a designated container. Based on our analysis, we enhance IMA to make a division of the traditional ML, for the purpose that a *verifier* can only collect the information of related subsets of ML, and hence the privacy issues can be solved. Our architecture also protects the integrity of these subsets by hardware-based *RoT*. The attestation mechanism is proposed as well. Key contributions in this paper are:

1. From the privacy's perspective, our architecture enables to attest a designated container without conveying other containers' information or unnecessary information of the underlying host to the remote *verifier*, in case of a malicious *verifier* being aware of which components the underlying host and other containers have.

2. From the security's perspective, our architecture enables to collect an integrity evidence, covering the designated container's all components and its dependencies. We present container-based PCR (cPCR) to ensure that the privacy-preserving integrity evidence is protected by a hardware-based *RoT*, and hence any manipulation to this evidence can be identified by the remote *verifier*.

3. We build a prototype based on TPM 1.2. Compared with the traditional IMA, the overhead incurred by Container-IMA is negligible, and the latency of Remote Attestation is even decreased. Besides, for the sake of usability, our architecture supports the current container-based architecture only with a minimum modification.

The rest of this paper is organized as follows. Section 2 reviews the traditional IMA. Section 3 introduces our motivation and overviews our architecture. Section 4 and Section 5 present the measurement and attestation mechanisms of our architecture respectively. The details of implementation and evaluation results are given in Section 6. Section 7 reviews some related works, and we conclude this paper in Section 8.

2 Integrity Measurement Architecture (IMA)

Trusted Computing dedicates to attesting the integrity of a remote platform. From the Trusted Computing's perspective, trust can be transferred step by step. The *CoT* is constructed from the *RoT* and extended with the entire boot time, including BIOS, GRUB, and finally the OS kernel. Any component in *CoT* is measured before loading, and finally a measurement evidence is generated for this *CoT*. As a part of *RoT*, TPM is viewed as trusted by default and implements necessary cryptographic algorithms. The PCRs shield in TPM are utilized to maintain a trusted evidence. Each TPM possesses 24 PCRs (from 0 to 23). The measurements of hardware, BIOS and bootloader stages are recorded into PCR0-7. Two operations for a PCR have been defined by TCG specifications [5]: one is *PCR_Reset* to reset it when the device powers on, and the other is *PCR_Extend*, which concatenates the old value of this PCR with the new measurement, and hashes the result as the new value. Therefore, once the platform starts up, all the values measured into a PCR cannot be reversed.

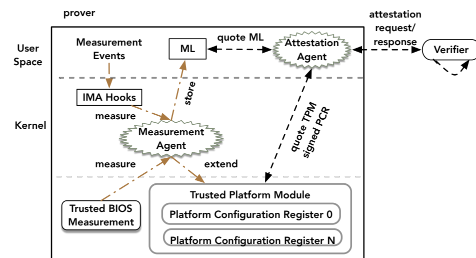


Figure 1: Integrity Measurement Architecture

Sailer et al. proposed IMA [46] to expand the scope of *CoT* to application layer. The architecture of IMA is shown in Figure 1. The IMA Hooks in *prover* inspect the measurement events, such as loading a binary program, *insmod*ing a kernel module or opening file by *root*. The Measurement Agent in *prover* is responsible for `measuring` a coming event. A *measure* means capturing this event's file path and calculating the hash value of the file content. The Measurement Agent

stores this *measure* into ML and extends it into PCR10 via *PCR_Extend*. ML is a supplement to PCR, since the size of PCR is fixed and it is impossible to recover the list of stored values backwards from the current content of a PCR. ML records the detailed list of the measurement values and necessary metadata for the software components, representing for the integrity status of the platform.

IMA has defined *Remote Attestation* to enable a *verifier* to validate the integrity of *prover*. When receiving an attestation request, the Attestation Agent in *prover* collects integrity evidence, including PCR values, the signature signed by TPM and the ML. This signature is well defined in TCG specifications [5], and the specifications ensure that the private key (i.e. Attestation Identity Key, AIK) can only be used inside a specific TPM attached to a specific platform. The PCR values are included in the signature. Hence, a valid signature represents for the identity of *prover* and the trustworthiness of the transferred PCR values. The genuineness of ML is further determined by simulating *PCR_Extend* and matching the simulated result with trusted value of PCR10. If the validation result is positive, the *verifier* searches his expected values and compares them with the trusted ML. The *verifier*'s expectations are stored into a Reference Manifest Database, which is established through collecting information from the original source: the software and hardware manufacturers.

However, transmitting the entire ML to *verifier* violates privacy in a container setting. A *verifier* can be the owner of a designated container. He should not obtain the unnecessary information of other containers and the underlying host. It is significant to ensure privacy during Remote Attestation.

3 Motivation and Architecture Overview

3.1 Scenario and Threat Model

Figure 2 shows a use case in a container setting. We define *prover* as the platform hosting quite a few containers for multiple users. For example, user A and user B each has two containers running in *prover*. *Verifier* is defined as a remote user, concerning about the integrity status of his containers running in *prover*. User A wants to know whether his container (e.g. mysql) is running as he expects, such as booting from a correct image and loading benign softwares.

Prover in Figure 2 shows a simplified container architecture. The `container management services`, e.g. Docker Daemon, are responsible for spawning and managing containers. All other components running in the underlying host are classified into `host applications`. When a client requests to run a container, the `container management services` locate and load the container image with specified configurations, such as configuring network and setting isolation environment. Linux `namespace` mechanism [33] is utilized for containers to establish isolation environments. This mechanism divides the system resources into many different instances in several aspects, including mount, hostname, IPC, PID and network [11]. Each container usually has a unique `namespace`, such that one container is isolated from others[1].

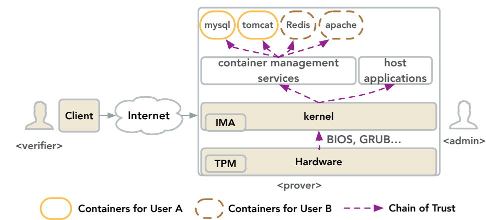


Figure 2: Use Case in a Container Setting

We assume that *prover* possesses trusted hardware, e.g. TPM, to support Trusted Computing, and `trusted boot` [35] ensures the integrity of OS kernel. We do not consider physical attacks and the attacker cannot get TPM's ownership. As for the capability of adversary, we focus on the *Local Adversary* and *Remote Adversary* [4]. A local adversary is sufficiently near *prover* to be capable of eavesdropping on, and interfering with, the *prover*'s communication. A remote adversary can remotely infect *prover* with malware, e.g. modifying files or integrity evidence a container relies on, affecting the attestation mechanism or even impersonating as `container management services`. Finally, we assume that only an authorized *verifier* can perform Remote Attestation, as a cluster should have an effective user management module.

Similar to other existing mechanisms based on Trusted Computing [9, 15, 22, 50], the run time memory attacks are not considered. This limitation can be mitigated by leveraging existing mechanisms, such as address space layout randomization [51] and control flow attestation [3], etc.

3.2 Notations

Measurement Event (ME): represents for an event which triggers a *measure*. MEs are defined by IMA through a policy. The default IMA policy measures all system sensitive files, including executables, mmapped libraries, and files opened for read by root [1]. Our work adheres to this definition.

Measurement Log (ML): each entry in ML represents for the measurement result and related metadata. Equation 1 shows details, where $measure(boot)$ refers to measure components in the *prover*'s boot time by `trusted boot` [35], and $measure(ME)$ is the result of measuring ME by IMA. At least the path and hash value of this ME are included into the measurement results. To protect each entry's integrity, IMA calculates the digest value of this entry, and appends this digest value ($node_{hash}$) into $measure(ME)$.

$$\begin{aligned} ML &= measure(boot) \cup \{measure(ME)\} \\ &= measure(boot) \cup \{< e.path, e.hash, e.node_{hash} > \mid e \in ME\} \end{aligned} \quad (1)$$

[1] Sharing namespaces among containers is discussed in Section 6.4.

3.3 Analysis

Traditionally, `trusted boot` [35] enables to measure components during a platform's boot time. With a *measure-before-loading* mechanism, a *CoT* is constructed and trust is transferred from *RoT* to the OS kernel. The measurement mechanism is changed when trust reaches to the application layer. All software components in the application layer are measured by IMA and written into ML according to the time of loading. Consequently, the sequence of loaded software components is conceived to the *CoT* in the application layer. The entire *CoT* can be decomposed into the following partitions:

1. *Integrity of prover's Boot Time* (I_{boot}^{Pro}): starts from powering on *prover* and ends with successfully loading OS kernel. It includes BIOS, GRUB and OS kernel.

2. *Integrity of Containers' Dependencies* (I_{dep}^{Con}): refers to the `container management services` and files or libraries they require.

3. *Integrity of a Container's Boot Time* (I_{boot}^{Con}): refers to the images and boot configurations when the `container management services` launch a container.

4. *Integrity of a Container's Applications* (I_{app}^{Con}): starts from `container management services` successfully launching a container and ends with shutting down it. It includes all components running inside a container.

5. *Integrity of Host Applications* (I_{app}^{Host}): starts from successfully launching OS kernel and ends with shutting down *prover*. The `container management services` and all containers are not included in this partition.

Figure 2 shows the containers' *CoT*. From a container's perspective, its direct dependency is the `container management service`. Other `host applications` are isolated from containers through namespaces. Hence a container's *CoT* only includes I_{boot}^{Pro}, I_{dep}^{Con}, this container's I_{boot}^{Con} and I_{app}^{Con}. When a *verifier* requests to attest a container, the *prover* can aggregate these partitions to convince its trustworthiness. And components which do not belong to the *CoT* of a designated container should not be revealed to a *verifier*.

Hence the privacy requirement considered in this paper is: *the integrity evidence, that the prover sends back to verifier, should not expose the information of* I_{app}^{Host}*, other containers'* I_{app}^{Con} *and* I_{boot}^{Con}. Note that a strong privacy requirement is that a *verifier* cannot distinguish whether his container is the only container running on *prover*. We will discuss it in Section 6.1.

To meet this privacy requirement, we need to divide the traditional ML into the above partitions. One of the challenges is how to make a division of ML automatically. Kernel should be empowered to know which partition a ME belongs to. Through adding additional items in the IMA kernel module, existing mechanisms [15, 50] can differentiate I_{app}^{Host} and I_{app}^{Con}. But they cannot handle I_{dep}^{Con} and I_{boot}^{Con}. It is also insufficient for mechanism based on *rkt* [22], since it can only record I_{boot}^{Con}.

Another challenge is how to ensure the integrity of ML's each partition with a hardware-based *RoT*. The traditional IMA constructs a single ML and binds this ML to PCR10. In a container setting, launching N containers introduces N more I_{app}^{Con}s. It is not feasible to adopt the traditional IMA's strategy, i.e. binding each ML partition to a unique PCR. So schemes [15, 50] adhering to the traditional IMA cannot protect I_{app}^{Host} and I_{app}^{Con} with hardware-based *RoT* separately.

3.4 Architecture Overview

Figure 3 depicts our architecture. Compared with the traditional IMA, our architecture introduces three additional components, including the Split Hook, the Namespace Parser and the container-PCR (cPCR) Module. Since the container-based virtualization utilizes the `namespace` mechanism to isolate system resources from others, each container should possess a unique namespace. Split Hook inspects the syscall to create a new namespace, and notifies kernel a event to split ML. The Namespace Parser extracts the namespace number of the current process to identify which partition the current ME belongs to. cPCR module is liable for protecting each ML partition with TPM by maintaining cPCRs. cPCR is a data structure in kernel simulating for PCR.

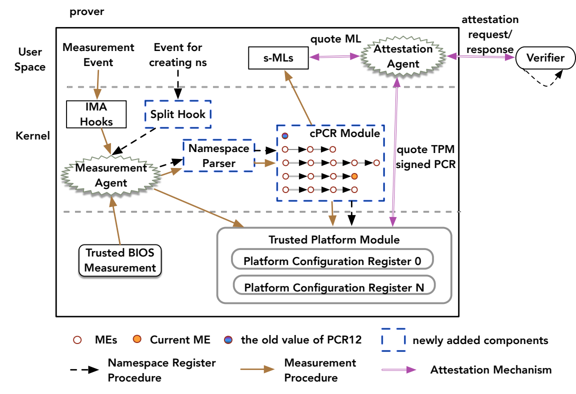


Figure 3: Overview Architecture of Container-IMA

Our architecture contains two parts: *measurement mechanism* (Section 4) and *attestation mechanism* (Section 5).

The *measurement mechanism* accomplishes its task with the help of these three newly added components. On one hand, Namespace Register Procedure contributes to splitting ML for each namespace. It starts from when the Split Hook is triggered and ends with creating and maintaining corresponding partition of ML and cPCR. On the other hand, Measurement Procedure is responsible for measuring MEs, storing them into corresponding partition of ML and protecting this partition with TPM via the assist of cPCR.

The *attestation mechanism* allows a remote *verifier* to request an integrity evidence of *prover*, for the purpose of

checking whether a designated container running as he expects. When receiving such a request, the modified Attestation Agent in *prover* locates the related ML partitions to ensure privacy, and retrieves the signed aggregated PCRs from TPM. Since the *measurement mechanism* ensures that all partitions are protected by physical PCRs, the trustworthiness of the received ML partitions can be determined.

4 Measurement Mechanism

This section elaborates the measurement mechanism. Note that components in the *prover*'s boot time, i.e. I_{boot}^{Pro}, are measured by `trusted boot` and protected by PCR0-7. They are inherently separated from other ML partitions. Hence we do not describe them in the remaining of this section.

4.1 Basic Namespace Register Procedure

Distinguishing different containers from the kernel's perspective is the basic problem we need to solve. Since a container should have a distinct namespace, kernel can distinguish containers through parsing the namespace number of the running process when performing the integrity measurement. And hence the I_{app}^{Host} and I_{app}^{Con}s can be separated.

Namespace Register Procedure is designed to empower kernel to make a division of ML. Kernel in our architecture has to maintain multiple double-linked lists to represent for the separated measurement logs (s-MLs) from ML. Equation 2 defines all s-MLs, where ME_{ns} refers to a ME whose namespace is *ns*, and *n* is the number of ML's partitions.

$$s\text{-}MLs = \{< value, ns >\}_n = \{< \{measure(ME_{ns})\}, ns >\}_n \tag{2}$$

In order to protect the integrity of these s-MLs, we simulated a set of PCRs, which we call as container-based PCRs (cPCRs). Each cPCR has its value, a unique *secret* and the corresponding namespace number, i.e. *ns*. *Secret* is a random value generated by TPM and is utilized to hidden other containers' cPCRs for a given *verifier* (See Section 4.2.1).

$$cPCR\text{-}list = \{cPCR\}_n = \{< value, ns, secret >\}_n \tag{3}$$

The workflow of Namespace Register Procedure is shown in Figure 3. When a container is launched, an event for creating namespace is generated. Split Hook captures the corresponding syscall, and the Measurement Agent knows this event and notifies Namespace Parser to parse the number of this new namespace, e.g. *ns*. Once receiving *ns*, cPCR Module will firstly request TPM to generate a random value as this cPCR's *secret*. Next, cPCR Module creates a new cPCR and a new s-ML, as the Equation 4 shows, where *AllZero* refers to the initialized value of a cPCR, i.e. all bytes are set to zero.

$$\begin{aligned} &cPCR\text{-}list := cPCR\text{-}list \cup \{AllZero, ns, secret\} \\ &s\text{-}MLs := s\text{-}MLs \cup \{< \{\}, ns >\} \end{aligned} \tag{4}$$

Namespace Register Procedure creates separated s-MLs for containers (namespaces), yet I_{dep}^{Con} and I_{boot}^{Con} are not divided. Section 4.3 presents the solutions.

4.2 Basic Measurement Procedure

The measurement procedure is responsible for measuring and recording MEs. Generating a ME triggers the IMA hooks. The Measurement Agent is notified and measures this ME. The measurement results ($measure(ME)$) and the namespace number of the current process (ns_{num}) parsed by Namespace Parser will be passed to the cPCR module. cPCR module afterwards locates the target cPCR and s-ML (Equation 5).

$$\begin{aligned} &target\text{-}cPCR = \{cPCR \mid cPCR.ns == ns_{num} \cap cPCR \in cPCR\text{-}list\} \\ &target\text{-}ml = \{s\text{-}ML \mid s\text{-}ML.ns == ns_{num} \cap s\text{-}ML \in s\text{-}MLs\} \end{aligned} \tag{5}$$

Secondly, the cPCR module extends the current ME into the target cPCR and appends it into the target s-ML (Equation 6). $ME.node_{hash}$ refers to the $node_{hash}$ in $measure(ME)$.

$$\begin{aligned} &target\text{-}cPCR.value := HASH(target\text{-}cPCR.value, ME.node_{hash}) \\ &target\text{-}ml.value := target\text{-}ml.value \cup measure(ME) \end{aligned} \tag{6}$$

The extend operation is the same as `PCR_Extend` provided in TPM specifications. The above steps ensure that the integrity of each s-ML is protected by the related cPCR, that is, we can check the genuineness of each s-ML by matching the related cPCR with the result of simulating Equation 6 with s-ML.

4.2.1 Bind cPCRs to Hardware-based *RoT*

After extending the target cPCR and appending the target s-ML, the cPCR module binds cPCRs into a physical PCR through the following steps.

1. records the history value of a specific PCR (*historyPCR*) which is not used in the current system. In our prototype based on TPM 1.2, we choose PCR12. We provide a GRUB parameter for user to change this PCR index.

2. records the digest of all cPCRs. *tempPCR* is initialized as $cPCR_0.value$ *xor*ed with $cPCR_0.secret$. For all other cPCRs in cPCR-list, Equation 7 is performed to extend them into *tempPCR*. Finally, *tempPCR* will save the digest value of all cPCRs' values.

$$\begin{aligned} &tempValue := cPCR_i.value\ xor\ cPCR_i.secret \\ &tempPCR := HASH(tempPCR \parallel tempValue) \end{aligned} \tag{7}$$

3. extends the physical PCR12 with the final *tempPCR*.

$$PCR12 := PCR_Extend(PCR12, tempPCR) \tag{8}$$

The *secret* field is essential to hidden other containers' cPCRs to a particular *verifier*. Given a user owning container

whose s-ML is extended into $cPCR_u$, a remote attestation means transferring a *nonce* to *prover* which is utilized to defend against replay attacks. The *prover* afterwards sends back at least the following values to the remote user (more details shown in Section 5): `historyPCR`, `sendcPCRs`, `Sign(AIK, nonce ‖ related PCRs)`, where *sendcPCRs* are values extended into PCR12 (Equation 9).

$$sendcPCRs = \{cPCR_i.value\ xor\ cPCR_i.secret \mid cPCR_i \in cPCR-list\} \tag{9}$$

In this case, the *verifier* can validate TPM's signature and afterwards get the genuine *nonce* and PCR12. Next, *verifier* validates *sendcPCRs*' genuineness via recalculating a simulated value, i.e. performing Equation 7 and extending *historyPCR* with the returned digest value. If the calculated result equals the decrypted PCR12, the trustworthiness of *sendcPCRs* is determined. Finally, if the *verifier* has obtained $cPCR_u.secret$, he can restore the $cPCR_u.value$ by *xor*ing $cPCR_u.secret$ with corresponding entry in *sendcPCRs*. In our current prototype, when a user successfully launches a container, this container's *secret* is fed back to user, and from that moment the *prover*'s kernel will not expose this *secret* to the user space. We will further discuss the relevant privacy issues in Section 6.1.

In summary, a one-by-one protection chain is established, i.e. TPM protects PCR12 which further protects cPCRs, and cPCRs protect s-MLs.

4.3 Extensions

Currently we haven't record the *Integrity of Containers' dependencies* (I_{dep}^{Con}) and *Integrity of Containers' Boot Time* (I_{boot}^{Con}) separately. The former does not create a new namespace, so the above mechanism fails to identify them. For the latter, a container's images and boot configurations are upper concepts in the user-space. Thus it is hard to let kernel parse and record them. This section gives our solutions.

4.3.1 Integrity of Containers' Dependencies (I_{dep}^{Con})

To reuse the Namespace Register Procedure, we add a new special program named `bootstrap program`, whose task is to create a new namespace for an application. Therefore, when launching these dependencies with `bootstrap program`, the Namespace Register Procedure will be triggered. Note that the container's dependencies are extended into $cPCR_0$, as they run ahead of any other containers. Besides, all containers rely on these dependencies, so we choose $cPCR_0.secret$ to be a well-known value, e.g. *AllZero*.

In our prototype, we choose `unshare` to be the `bootstrap program`, because `Docker-ce` [16] utilizes the unshare syscall to allocate a new namespace for a container. We have to enable the *verifier* to check: ① the integrity of `bootstrap program`, and ② whether his containers are indeed launched with services generated by `bootstrap program`.

The former issue can be addressed by adding the process which creates new namespace (hereafter we call this process as `createProcess`) into the related s-ML. For instance, the `createProcess` of containers' dependencies refers to `bootstrap program`. Since s-MLs are finally protected by PCR, the integrity of `bootstrap program` is protected as well. Thus, Equation 4 should be modified to:

$$\begin{aligned} &cPCR\text{-}list := cPCR\text{-}list \cup \\ &\quad \{OP_{extend}(AllZero, measure(createProcess)), ns, secret\} \\ &s\text{-}MLs := s\text{-}MLs \cup \{< \{measure(createProcess)\}, ns >\} \end{aligned} \tag{10}$$

where $OP_{extend}(AllZero, measure(createProcess))$ indicates that this cPCR is extended with the measurement results of *createProcess* with an initialized value *AllZero*.

A simple way to deal with the latter issue is to record `pid` of `createProcess` and its ancestor processes' `pids`. We define these `pids` as `createProcess.pids`. If a container is launched with the genuine dependencies, the `pid` of dependencies' `createProcess` can be found in `pids` of this container. And hence a *verifier* can determine whether the s-ML for containers' dependencies is genuine. The measurement of `createProcess`, i.e. P, is changed to:

$$measure(P) = < P.path, P.hash, P.pids, node_{hash} > \tag{11}$$

Figure 4 gives an example. File *ascii_runtime_measurements* records I_{app}^{Host}, which is irrelevant to containers. File 4026532222 and 4026532238 refer to s-MLs for containers' dependencies (I_{dep}^{Con}) and the launched container (I_{app}^{Con}), respectively. Each entry of s-ML comprehends four elements: `index` (pcr index or namespace number), `template-hash` (i.e. $node_{hash}$), `template-name`, `file-hash` (the digest value for measured file) and `file-path`. We omit the `template-hash` in Figure 4. For instance, the second entry in file 4026532222 means that the measured file is `/usr/bin/dockerd` in namespace 4026532222.

In Figure 4, the first entries in file 4026532222 and 4026532238 refer to the corresponding `createProcess`. It means that containers' dependencies are launched by `/usr/bin/unshare` whose pid is 1990. The container is launched by `/usr/local/sbin/runc` whose pid is 2358 and it is a child of process 1990. Therefore, file 4026532222 indeed represents for this container's dependencies.

4.3.2 Integrity of All Containers' Boot Time (I_{boot}^{Con})

The integrity of containers' boot time includes the image, configuration and parameters to bootstrap a container. These information cannot be obtained by kernel as they are concepts in application layer. Considering that the containers' dependencies have been genuinely recorded and protected by *RoT*, we can let them collect this information.

In our prototype, we modify `runc` [2] to do the measurement and extend $node_{hash}$ into another physical PCR, e.g.

```
==> 4026532222 <==
4026532222 ... ima-ng sha1:38919a201521117fb8bf907ab5d41bb31eb29a39 1990->1980->1907->1447->1249->1071->1->0_4026532222:/usr/bin/unshare
4026532222 ... ima-ng sha1:a348d30d0774ed4d84945da0f10d6319da4cd9ac 4026532222:/usr/bin/dockerd
4026532222 ... ima-ng sha1:80a5ea753fe06e9ecc8f5bdf857d4af9d8aef39b 4026532222:/usr/bin/docker-containerd
4026532222 ... ima-ng sha1:126ee56dd59433f8a488cf873d0fe6fea2c3f91a 4026532222:/var/lib/docker/tmp/docker-default618280113
4026532222 ... ima-ng sha1:a00d3061edf0ff271e7e7395d2d9676736f95477 4026532222:/lib/modules/3.13.11-ckt39/kernel/ubuntu/aufs/aufs.ko

==> 4026532238 <==
4026532238 ... ima-ng sha1:d5442f82ce6ee98f17b4a21e49fdd326e4d1c6ae 2358->2354->2346->2001->1990->1980->1907->1447->1249->1071->1->0_4026532238:/usr/local/sbin/runc
4026532238 ... ima-ng sha1:611a59c515074dbb376713fd19040c10a0c8e5e2 4026532238:/bin/bash
4026532238 ... ima-ng sha1:b43aec6bd95cab18f2bcedf1dfcf01462f7e088d 4026532238:/lib/x86_64-linux-gnu/ld-2.23.so
4026532238 ... ima-ng sha1:eaae87c3d507be4634db12ab562c9f1cb243e764 4026532238:/etc/ld.so.cache
4026532238 ... ima-ng sha1:2bd9389f52439de7a42efcb8a3c3f8d4c881e8b2 4026532238:/lib/x86_64-linux-gnu/libtinfo.so.5.9

==> ascii_runtime_measurements <==
10 ... ima-ng sha1:27d6a1e1108a90c19003d919f325c7bd0f4acb10 boot_aggregate
10 ... ima-ng sha1:a5e65f1ca3a779cea954a015bde7ec5daa3f7612 4026531840:/init
10 ... ima-ng sha1:dc3e621c72cde19593c42a7703e143fd3dad5320 4026531840:/bin/sh
10 ... ima-ng sha1:67c253d8ea7089253719cad7f952fb4c22240f27 4026531840:/lib64/ld-linux-x86-64.so.2
10 ... ima-ng sha1:fac553d7706114a2ed0ef587aaf48f58e19f381a 4026531840:/etc/ld.so.cache

==> docker-boot <==
"... [4026532238] sha256:7aa3602ab41ea3384904197455e66f6435cb0261bd62a06db1d8e76cb8960c42 ... /var/lib/docker/containers/.../config.v2.json" c952b062e8be3cbc407242cb2ebcb27c8111b489
```

Figure 4: An example for the first 5 entries of each ML partition after we bootstrapped `dockerd` (Docker Daemon) through `unshare` command and afterwards set up a new container via `docker run -ti ubuntu:16.04`.

PCR11. `Runc` is responsible for spawning and running containers. File `docker-boot` in Figure 4 gives an example for s-ML related to container's boot time. Each entry in this file is composed of six elements: `id` (the container's id), `ns` (the namespace number), `HASH(image)` (the digest value of its image), `HASH(config)` (the digest value of its configuration), `PATH(config)` (the absolute path of its configuration file), and $node_{hash}$. In Figure 4, we replace the `id` and `HASH(config)` with symbol $'...'$ to have a better display.

One privacy issue is that we cannot transfer the entire `docker-boot` to *verifier* because it records all container's boot information. Instead, we only transfer the $node_{hash}$ for other containers in our prototype.

5 Attestation Mechanism

This section shows how to enable a remote *verifier* attesting the integrity of a designated container and its dependencies. Our attestation mechanism derives from the traditional Remote Attestation. Note that we assume that there is an effective user management system in a cluster, e.g. kubernetes [32], to prevent unauthorized *verifier* from launching attestation mechanism. Hence we do not consider identifying *verifier*s.

Message Transferring When verifying the integrity of a container running in *prover*, a remote *verifier* should send a `request: <nonce, containerID>`, where *nonce* is a random number generated by *verifier*, and *containerID* refers to the target container. When receiving this `request`, the Attestation Agent in the *prover* collects:

1. the related PCR values and the signature of TPM via performing *TPM_Quote*, i.e. $Sign\{AIK, nonce \parallel PCRs\}$, where PCRs refer to PCR0-7, PCR11 and PCR12.

2. *sendcPCRs* (See Equation 9) and the history value of PCR12 (i.e. *historyPCR*).

3. s-ML for *prover*'s boot time and the containers' dependencies, i.e. I^{Pro}_{boot} and I^{Con}_{dep}.

4. s-ML for containers' boot time components (I^{Con}_{boot}), which is described in Section 4.3.2. For the target container, the entry in this s-ML includes `id, ns, HASH(image), HASH(config), PATH(config) and` $node_{hash}$. For other containers, it only contains $node_{hash}$.

5. s-ML for the target container's applications, i.e. I^{Con}_{app}.

Workflow of *Verifier* Firstly, *verifier* validates the identity of TPM. Each TPM is written a unique endorsement certification by manufacture. The endorsement key (EK) can be used to identify a TPM. AIK is constructed from EK and generates signatures. Matching the received PCR values with a correct TPM signature confirms the correctness of PCR values. The trustworthiness of *nonce* can be determined as well, and *verifier* knows the freshness of this response.

Secondly, based on PCR values which are determined as trusted, the integrity of `s-MLs` for bootstrapping *prover* (I^{Pro}_{boot}) and the target container (I^{Con}_{boot}) can be confirmed. These `s-MLs` are extended into physical PCRs, i.e. PCR0-7 for I^{Pro}_{boot} and PCR11 for I^{Con}_{boot}. The genuineness of these `s-MLs` can be checked by simulating the `PCR_Extend` operation and comparing the simulation result with the trusted PCRs.

Thirdly, the trusted PCR12 can be used to verify the genuineness of *sendcPCRs*. The *verifier* calculates *tempPCR* through Equation 7 with the received *sendcPCRs*. Afterwards, a simulated PCR value (*sPCR*) can be calculated by extending the *historyPCR* with *tempPCR*. If the final *sPCR* equals to PCR12, *verifier* gets a trusted *sendcPCRs*, otherwise some attacks have happened. With the genuine *sendcPCRs* and the *secret*, *verifier* can get his concerned container's cPCR (e.g. $cPCR_u.value$) and $cPCR_0.value$.

Finally, the trustworthiness of I^{Con}_{app} and I^{Con}_{dep} can be determined by simulating `PCR_Extend` and matching the results with $cPCR_u.value$ and $cPCR_0.value$, respectively. The *veri-*

fier should also confirm whether I_{dep}^{Con} is indeed his container's dependency, as we mentioned in Section 4.3.1.

If all aforementioned validation procedures are positive, all related `s-MLs` are trusted. Then the *verifier* can validate `s-MLs` against his expectations. These expectations are defined as the same as the traditional Trusted Computing, which are collected from the software and hardware manufacturers.

6 Implementation and Evaluation

This section presents our implementation details, analyzes the privacy and security achievements, and illustrates the experimental results. A prototype is implemented to evaluate our solution. The *prover* runs Ubuntu 14.04 with our modified kernel based on kernel 3.13.11 to support Measurement Mechanism, along with 16GB memory, 4 processor cores. A TPM1.2 chip is equipped in *prover*. Docker-ce [16] is installed in *prover* with version 17.06.1-ce. The *verifier* is Ubuntu 14.04 with the default kernel and 16GB memory.

Measurement Mechanism requires to hook action which creates a new namespace. We hook the `unshare` syscall to perform Namespace Register Procedure. Meanwhile, if a ME does not have a namespace number or its namespace number can not be found in `s-MLs`, this event belongs to `host applications` in *prover*. In this case, the measurement result is extended into PCR10 as traditional IMA does.

The implementation of Attestation Mechanism is based on the OpenAttestation (OAT) [27] project, which is an open-source project to support PCR-based report schema. We modified the message transferring protocol and the validation procedure of OAT to support our Attestation Mechanism.

6.1 Privacy

The privacy achievements comprise two aspects. Firstly, the `host applications`' information (i.e. I_{app}^{Host}) are not necessarily sent back to *verifier*. In our solution, the s-ML related to `host applications` are recorded into a separate file. This s-ML is extended into PCR10 as traditional IMA. Both this s-ML and PCR10 are not required during the remote attestation for containers. Secondly, other container's boot time information (i.e. I_{boot}^{Con}) and applications (i.e. I_{app}^{Con}) are not exposed. Although all containers' boot time are recorded into the same s-ML, the information transferred to *verifier* only contains the corresponding $node_{hash}$. Since the container *id* is a random value, the *verifier* cannot obtain meaningful information from $node_{hash}$. We can even add more unguessable value into I_{boot}^{Con} to confuse the malicious *verifier*. Meanwhile, other containers' I_{app}^{Con}s are not transferred for remote attestation.

A privacy issue is related to *cPCR.value*. A standardized container has a well-known boot hash. Knowing *cPCR.value* may allow the attacker to deduce which container runs. In our solution, other containers' *cPCR.value*s are *xor*ed with corresponding *cPCR.secret*s. The results of *xor* operation (i.e. *sendcPCRs*) are sent to *verifier*. Since other containers' *cPCR.secret*s are unknown to a *verifier*, he cannot restore their *cPCR.value*s. An issue is how to ensure the privacy of *cPCR.secret*. In our prototype, we feed back *secret* to user once a container is successfully launched and afterwards ensure that *secret* will not be exposed to the user space. This strategy requires a trusted communication between user and *prover* at this moment, which we think is reasonable in a container cluster. An alternative strategy is to let *prover* generate a key pair and bind the private part (*sk*) into TPM in the initialization procedure. All users obtain the public part (*pk*) and perform a remote attestation by encrypting *nonce* with *pk*. Since only *prover*'s TPM can get *sk*, *nonce* cannot be leaked to attackers. Afterwards, *prover* can *xor nonce* with the target *secret*. The *xor*ed result is utilized as the previous remote attestation's nonce to get TPM's quote. In this case, *verifier* can obtain a genuine *nonce xor secret* through attestation response. By *xor*ing with the *nonce* the *verifier* sent before, the *secret* can be restored. A malicious *verifier* cannot obtain other users' *secret*s without *nonce*s, so the privacy of *secret*s is ensured. We will implement this strategy in the future.

Finally, as we mentioned in Section 3.3, a much stronger privacy requirement is that any *verifier* cannot distinguish whether his container is the only container running on *prover*. Having this knowledge helps an attacker decide whether to launch a co-resident attack [21]. In Container-IMA, a malicious *verifier* can get the current number of containers running on *prover* by counting the size of *sendcPCRs*. We can use some obfuscated strategies to solve this problem. For instance, we can add a GRUB parameter to configure *prover* to run at most *max* containers. The unused cPCRs are filled with unmeaningful values. Since all *xor*ed cPCRs are sent to *verifier*, the size of *sendcPCRs* is not a privacy sensitive data. Another similar problem is that the attacker can perform remote attestation several times to inspect *historyPCR*. If this value is changed, attacker knows that another container performs some operations. We can let *prover* update PCR12 when a *host application*'s ME occurs. This operation confuses the *verifier* whether the change of *historyPCR* is caused by host applications or containers' events. Considering loading additional host applications may be rare after a *prover* is deployed, we can let kernel do some timely unaffected measures and update PCR12. In summary, although our prototype does not enforce obfuscated strategies, it is feasible to solve these issues.

6.2 Security

The security achievements include two aspects: the integrity evidence ① covers a container's entire *CoT*, and ② is protected by hardware-based *RoT*. Container-IMA achieves ML partition and protection for each partition. The I_{boot}^{Pro} is measured by `trusted boot` and is recorded into PCR0-7. The I_{dep}^{Con} and I_{app}^{Con} are recorded into cPCRs, and are further protected by PCR12. The I_{boot}^{Con} is extended into PCR11. Hence the

evidence is well protected and its completeness is achieved. Note that the mechanism based on *rkt* [22] does not cover the entire *CoT*, as it only considers the container's boot phase. Similarly, it is not enough to ensure container security by simply blacklisting of known vulnerable container images.

Compared with integrity mechanisms based on vTPM, our approach does not require an additional layer in the `user space`. The typical integrity mechanisms to leverage vTPM are [26] and [9]. The former presents two possible architectures to build trust for containers through vTPM. One architecture is to put the vTPM instances in the host OS kernel. The container manager in the user-space must be responsible for asking the kernel to create a new vTPM and assigning the device to a container. Another architecture is to put vTPM and vTPM manager in one of the privileged container, which is similar to Dom0 in Xen [8]. Both these two architectures require a user-space component to bind vTPMs into physical TPM. The latter mechanism [9] implements a vtpm proxy driver in the Linux kernel that enables to spawn a TPM device with a client TPM character device and a server side file descriptor. The client device is moved into the container by creating a character device, while the server side file descriptor is passed to the TPM emulator in the user space. All these mechanisms require additional trusted components in the user space. Once these components are affected by attacker, the integrity evidence is not protected. Our solution relies on cPCR module, but it is inside the OS kernel, and the integrity of kernel is measured by `trusted boot`. Finally, all these two mechanisms do not elaborate the measurement and attestation mechanisms for containers.

6.3 Efficiency

6.3.1 Performance of Containers

This section shows our evaluation on the performance of containers. In each case, we test two kernels: One is `Default` referring to the default kernel 3.13.11 with IMA enabled, the other is `Container-IMA` which implements our Measurement Mechanism based on the default kernel 3.13.11. IMA is enabled by adding `ima_tcb` in GRUB command line.

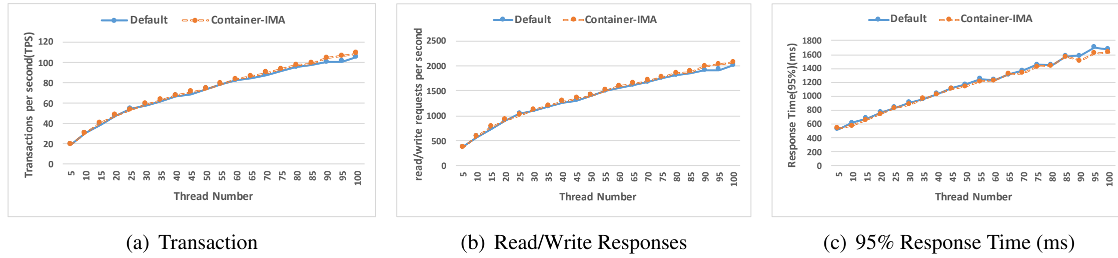
Influence on Basic Environment We run a Ubuntu 16.04 container in *prover* to evaluate the overhead of our modified kernel. The benchmark tool is `LMbench3` [36], a well-known tool for performance analysis. For each kernel, we run `LMbench3` for 20 times and record their average usages. Table 1 shows the experiment results. The percent overhead calculation against case `Default` is shown in case `Container-IMA`. As Table 1 shows, for most measurements, the evaluation results of case `Container-IMA` cost a little more than case `Default`. It is reasonable, since a container's ME leads to extending cPCR and binding cPCRs into physical PCR, while the `Default` kernel just extends PCR10.

Table 1: Influence on Container

Type	Default	Container-IMA
Processor, Processes (microseconds) - smaller is better		
null call	0.199	0.199(0.00%)
null I/O	0.27	0.27(0.00%)
stat	2.311	2.328(0.74%)
signal install	0.2675	0.267(-0.19%)
signal handle	1.04	1.04(0.00%)
exec process	371.5	374.85(0.90%)
sh process	1510.4	1514.1(0.24%)
Local Communication bandwidths in MB/s - bigger is better		
Pipe	4518.25	4503.45(-0.33%)
AF Unix	13.8K	13.9K(0.72%)
File reread	9150.215	9148.97(-0.01%)
Bcopy(libc)	10.K5	10.5K(0.00%)
Bcopy(hand)	7566.39	7562.99(-0.04%)
Mem read	14K	14K(0.00%)
Mem write	11K	11K(0.00%)

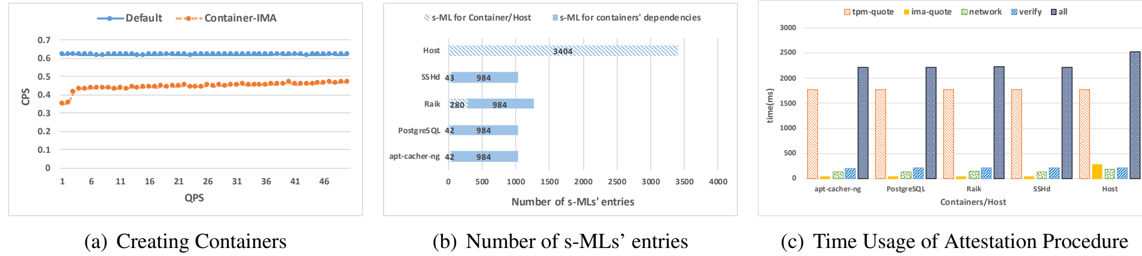
Influence on Container Applications In this experiment, we choose MySQL to be the tested application. We run a MySQL container and utilize `Sysbench` [29] to create a test table with 2,000,000 rows of data. We increase the number of used threads from 5 to 100 with the step of 5. In each case we record the transactions per second (TPS), the read/write responses per second (RPS) and the average response time for more than 95% requests. The experiment results are shown in Figure 5(a), 5(b) and 5(c) respectively. When the number of threads increases, TPS, RPS and the response time tend to be increased. Compared with case `Default`, the influence of our `Container-IMA` kernel is negligible. The default IMA policy measures the binary programs, the dynamic link libraries, the kernel modules and files opened by `root`. Operating MySQL does not trigger much of these measurements. Note that most containers are developed as designated applications. These containers trigger little measurements once they become stable, such that the introduced overhead is negligible.

Influence on Spawning Containers Kubernetes provides `Docker Micro Benchmark` [30] to evaluate Docker operations. We utilize this tool to benchmark the overhead of spawning containers via `benchmark.sh -o`. This tool launches 100 *go routines* keep starting containers until the latency is larger than 50s. We gradually increase the Queries Per Second (`QPS`) and record the related Container Started Per Second (`CPS`). `QPS` represents for the limited query rate of all *go routines*. Figure 6(a) shows the results. For example, when `QPS` is set to 20, our solution creates 0.4448 containers per second, while Docker Daemon running in the default kernel creates 0.6185 containers per second. In case `Container-IMA`, creating a container means creating cPCR and s-ML, measuring the binary program, extending cPCR and binding cPCRs into physical PCR. It is reasonable that our solution introduces some overhead towards spawning containers.



(a) Transaction (b) Read/Write Responses (c) 95% Response Time (ms)

Figure 5: Experiments for MySQL Running in a Container



(a) Creating Containers (b) Number of s-MLs' entries (c) Time Usage of Attestation Procedure

Figure 6: Experiments for Spawning Containers and Attestation Procedure

6.3.2 Performance of Attestation Procedure

Docker community gives some sample applications [17] to show how to run popular software using Docker. We choose a part of them to evaluate our solution, including *apt-cacher-ng*, *PostgreSQL*, *Riak* and *SSHd*. After starting up *prover* and launching above sample containers, we firstly record the number of MEs for each s-MLs. Figure 6(b) shows the details, where label `Host` refers to the s-ML for *host applications*, and other labels of the vertical axis refer to the s-MLs of sample containers. For example, container `Riak` possesses 280 MEs for its run time components and 984 MEs for its dependencies, while the *Host* has 3404 MEs for *host applications*.

We then test the cost of attestation. For each case, we record the time usage within 20 minutes and calculate their average value for five phases: `tpm-quote` (*prover* collects signature for PCRs via `TPM_Quote`), `ima-quote` (*prover* collects other evidence except PCRs' signatures), `network` (transferring attestation request and integrity evidence), `verify` (the *verifier* validates the trustworthiness of integrity evidence), and `all` (the whole attestation procedure). Figure 6(c) shows the experiment results. For example, with respect to *PostgreSQL*, the average time for these five phases are 1768.76ms, 52.72ms, 141.50ms, 214.78ms, 2217.33ms respectively.

The time usages for attesting containers are nearly in the same order. During the attestation procedure, the *prover* collects *sendcPCRs*, s-MLs for this container, containers' dependencies and the signature of TPM for PCRs. The size of *sendcPCRs* and the number of quoted PCRs are the same value for attesting containers. And the number of a container's s-MLs, which is shown in Figure 6(b), is also nearly the same order of magnitude. Consequently, attesting them takes nearly the same time. Attesting *Host* differs from containers. It does not require to collect and validate cPCRs. However, since the number of MEs for *Host* is much higher than a container's, it takes more time to collect (phase `ima-quote`) and transfer (phase `network`) these MEs. Hence attesting *Host* requires more time than attesting a designated container.

Note that the traditional IMA records all MEs into a single ML, including the *Host*'s and containers'. When a *verifier* attests a container, the *prover* should response with all MEs, which will be larger than a container's s-MLs in our solution. Because of the larger size of MEs, the cost of attesting a *prover* through the traditional IMA is larger than our solution.

6.4 Discussion and Limitation

During a container's life cycle, the user may need to stop/restart or pause/unpause it for the sake of usability. These operations do not change the container's namespace, such that Container-IMA is able to support them. However, a container may be migrated in a cluster. Container-IMA does not cover the container migration. Solving this problem will be our future work.

Besides, containers are able to share namespaces in some container clusters. For example, Kubernetes provides Pod for containers with shared namespaces and volumes [31]. A Pod usually hosts containers which are relatively tightly coupled, meaning that these containers commonly need to be considered as a whole. Although Container-IMA does not separate s-MLs for them, transferring these s-MLs does not hinder the privacy. Additionally, a sophisticated attacker may run his container in the same Pod as the victim's container, e.g. through tricking the cluster scheduler. In this case, the attacker's operations will be recorded into the same s-ML

with the victim container. The victim user can be aware of the attacker's existence through the remote attestation. However, if a container cluster itself improperly schedules containers from different users with shared namespaces, there should be a privacy breach. We will consider this problem in the future.

7 Related Work

Trusted Computing Trusted Computing contributes to verifying the trustworthiness of *prover*. Based on IMA, binary remote attestation enables a *verifier* to determine the current integrity state of *prover*. Stumpf et al. [48] propose the Timestamped Hashchain-based Attestation to compensate the deficiency of remote attestation. Considering the large cost of transmitting and calculating ML, Jaeger et al. [28] present Policy-Reduced IMA (PRIMA) which leverages the information flow and SELinux to measure the code and data related to the target application. This limits the scope of the attestation target applications, and hence reduces the size of ML. Some other works [34, 41] also achieve ML reduction.

Virtual TPM (vTPM) [10] devotes to building trust for the hypervisor-based virtualization environment. Through constructing multiple separated *RoTs* for VMs by vTPM, Remote Attestation works for VMs. Several types of vTPM are concluded in [52], such as software-based vTPM [10], hardware-based vTPM [47], para-vTPM [19] and property-based vTPM [45]. Additionally, TCG group has released the TPM 2.0 specification [24]. Chen et al. [13] present a new digital signature primitive in TPM 2.0 with provable security. Raj et al. [40] describe a software-only implementation of TPM 2.0 used in millions of mobile devices. Camenisch et al. [12] research on the TPM 2.0 interfaces and propose a revision to obtain better security and privacy guarantees which requires only minimal changes to the current TPM 2.0 commands.

Container Security Container-based virtualization offers a light-weight virtualization approach to run multiple environments using the host kernel, yet it faces many security issues. Some typical usages of Docker are presented in [14], and the authors discuss Docker's security implications and point out its vulnerabilities. Various mechanisms have been proposed to enhance the security of Docker. The whitepaper of NCC Group [23] explores various security mechanisms for containers, and enumerates strong security recommendations to counter deployment weaknesses. Harbormaster [54] is proposed to enforce policy checks and implement the principle of least privilege for Docker.

Substantial efforts are made to measure the integrity status of containers and then build trust for containers. The mainstream mechanisms are based on the following technologies: vTPM, IMA, *rkt* and SGX.

Some researchers propose solutions based on vTPM [9, 26]. Hosseinzadeh et al. [26] present two possible architectures to build trust for containers through vTPM. Both these two architectures require a user-space component to bind vTPMs into physical TPM. Berger et al. [9] implement a vtpm proxy driver in the Linux kernel that enables to spawn a TPM device with a client TPM character device and a server side file descriptor. The server side file descriptor is passed to the TPM emulator in the user space. Compared with them, our approach does not require an additional layer in the `user space` to manage and coordinate multiple vTPM instances.

Leveraging IMA is another way to build trust for containers. IMA utilizes the hash value of binary code to identify a software component, and records its file path and hash value to ML. Meanwhile, container-based virtualization, e.g. Docker, introduces the `mount namespace` to isolate containers. Different containers may have some files with the same path, e.g. */bin/ls*, yet IMA regards them as the same software component. To address this problem, researchers in [15, 50] modified IMA to add an additionally item indicating which container the process belongs to, thus it empowers IMA to measure the integrity of containers. However, each *verifier* has to collect the entire ML when validating the integrity status of *prover*, regardless of a container or the underlying host. The privacy requirements thus cannot be achieved. Sun et al. [49] propose `security namespace` for IMA, which allows containers to have their own MLs and IMA policies. The IMA policy is able to specify which PCR the ML is extended into [44]. Once the number of containers is bigger than the number of PCRs, some containers will share a same PCR. And these containers' MLs should all be sent back to *verifier* during the Remote Attestation. The privacy issue still exist. In our work, introducing cPCR is capable of solving this problem. Besides, they did not consider the integrity of container's dependencies.

Based on `rkt`, CoreOS team presents a mechanism [22] to measure the boot phase of containers, such as images and configurations, yet the components loaded in the run time of containers and their dependencies are ignored. And hence it is not enough to meet the security requirement. SCONE [7] leverage SGX trusted execution support to protect container processes from outside attacks. However, the performance overhead is more than expected with regards to the service running in native container environment.

8 Conclusion

In this work, we presented Container-IMA to build trust in a container setting. We firstly analyzed the essential evidence to validate the integrity of a designated container from the perspective of a remote *verifier*. Based on the analysis, we described how the kernel achieves ML partition to ensure privacy and how to utilize cPCR to protect each partition with a hardware-based *RoT*. The corresponding attestation mechanism is proposed as well. Our prototype and the evaluation results demonstrate that our architecture can satisfy the privacy, security and efficiency requirements.

Acknowledgments

This work was supported by National Natural Science Foundation of China under Grant No. 61672062 and No. 61232005.

References

[1] Integrity measurement architecture (ima). https://sourceforge.net/p/linux-ima/wiki/Home/.

[2] Runc. https://github.com/opencontainers/runc.

[3] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-FLAT: control-flow attestation for embedded systems software. In *ACM Conference on Computer and Communications Security*, pages 743–754. ACM, 2016.

[4] Tigist Abera, N Asokan, Lucas Davi, Farinaz Koushanfar, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. Invited-things, trouble, trust: on building trust in iot systems. In *Proceedings of the 53rd Annual Design Automation Conference*, page 121. ACM, 2016.

[5] Trusted Computing Platform Alliance. Main specification. *Version*, 1:1–284, 2000.

[6] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, 2013.

[7] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, et al. Scone: Secure linux containers with intel sgx. In *OSDI*, pages 689–703, 2016.

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 164–177. ACM, 2003.

[9] Stefan Berger. Virtual tpm proxy driver for linux containers. https://www.kernel.org/doc/html/v4.10/security/tpm/index.html, 2016.

[10] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vtpm: Virtualizing the trusted platform module. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[11] Thanh Bui. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015.

[12] Jan Camenisch, Liqun Chen, Manu Drijvers, Anja Lehmann, David Novick, and Rainer Urian. One tpm to bind them all: fixing tpm2. 0 for provably secure anonymous attestation. *Proceedings of IEEE S&P 2017*, 2017.

[13] Liqun Chen and Jiangtao Li. Flexible and scalable digital signatures in tpm 2.0. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 37–48. ACM, 2013.

[14] Theo Combe, Antony Martin, and Roberto Di Pietro. To docker or not to docker: A security perspective. *IEEE Cloud Computing*, 3(5):54–62, 2016.

[15] Marco De Benedictis and Antonio Lioy. Integrity verification of docker containers for a lightweight cloud environment. *Future Generation Computer Systems*, 97:236–246, 2019.

[16] Docker. Docker-ce. https://docs.docker.com/install/linux/docker-ce/ubuntu/.

[17] Docker. Sample applications for docker. https://docs.docker.com/samples/.

[18] Docker. Docker containers. http://docker.com/, 2016.

[19] Paul England and Jork Loeser. Para-virtualized tpm sharing. *Trusted Computing-Challenges and Applications*, pages 119–132, 2008.

[20] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[21] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. Containerleaks: Emerging security threats of information leakages in container clouds. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 237–248. IEEE, 2017.

[22] Matthew Garrett. What trusted computing means to users of coreos and beyond. https://coreos.com/blog/coreos-trusted-computing.html, 2015.

[23] Aaron Grattafiori. Understanding and hardening linux containers. *Whitepaper, NCC Group*, 2016.

[24] Trusted Computing Group. Tpm 2.0 library specification. https://trustedcomputinggroup.org/resource/tpm-library-specification/.

[25] Matt Helsley. Lxc: Linux container tools. *IBM devloperWorks Technical Library*, page 11, 2009.

[26] Shohreh Hosseinzadeh, Samuel Laurén, and Ville Leppänen. Security in container-based virtualization through vtpm. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pages 214–219. ACM, 2016.

[27] Intel. Openattestation (oat). https://01.org/OpenAttestation, 2014.

[28] Trent Jaeger, Reiner Sailer, and Umesh Shankar. Prima: policy-reduced integrity measurement architecture. In *Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 19–28. ACM, 2006.

[29] Alexey Kopytov. Sysbench manual. *MySQL AB*, pages 2–3, 2012.

[30] Kubernetes. Docker micro benchmark. https://github.com/kubernetes/contrib/tree/master/docker-micro-benchmark.

[31] Kubernetes. Sharing namespaces in pod. https://kubernetes.io/docs/concepts/workloads/pods/pod/.

[32] Kubernetes. Kubernetes. http://kubernetes.io/, 2016.

[33] Linux. Linux namespace mechanism. http://man7.org/linux/man-pages/man7/namespaces.7.html.

[34] Wu Luo, Wei Liu, Yang Luo, Anbang Ruan, Qingni Shen, and Zhonghai Wu. Partial attestation: Towards cost-effective and privacy-preserving remote attestations. In *2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, August 23-26, 2016*, pages 152–159. IEEE, 2016.

[35] Richard Maliszewski, Ning Sun, Shane Wang, Jimmy Wei, and Ren Qiaowei. Trusted boot (tboot), 2015.

[36] Larry W. McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, 1996.

[37] Chris Mitchell. *Trusted computing*, volume 6. Iet, 2005.

[38] Pradeep Padala, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, Kang G Shin, et al. Performance evaluation of virtualization technologies for server consolidation. *HP Labs Tec. Report*, 2007.

[39] A Polvi. Coreos is building a container runtime, rkt. *Retrieved*, 4:2015, 2014.

[40] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, et al. ftpm: A software-only implementation of a tpm chip. In *USENIX Security Symposium*, pages 841–856, 2016.

[41] Tobias Rauter, Andrea Höller, Nermin Kajtazovic, and Christian Kreiner. Privilege-based remote attestation: Towards integrity assurance for lightweight clients. In *Proceedings of the 1st ACM Workshop on IoT Privacy, Trust, and Security*, pages 3–9. ACM, 2015.

[42] Nathan Regola and Jean-Christophe Ducom. Recommendations for virtualization technologies in high performance computing. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 409–416. IEEE, 2010.

[43] Elena Reshetova, Janne Karhunen, Thomas Nyman, and N Asokan. Security of os-level virtualization technologies. In *Nordic Conference on Secure IT Systems*, pages 77–93. Springer, 2014.

[44] Eric Richter. Specifying pcr in the ima policy. https://git.kernel.org/pub/scm/linux/kernel/git/zohar/linux-integrity.git/commit/security/integrity/ima?h=next-namespacing-experimental&id=0260643ce8047d2a58f76222d09f161149622465.

[45] Ahmad-Reza Sadeghi, Christian Stüble, and Marcel Winandy. Property-based tpm virtualization. *Information Security*, pages 1–16, 2008.

[46] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *USENIX Security Symposium*, volume 13, pages 223–238, 2004.

[47] Frederic Stumpf and Claudia Eckert. Enhancing trusted platform modules with hardware-based virtualization techniques. In *Emerging Security Information, Systems and Technologies, 2008. SECURWARE'08. Second International Conference on*, pages 1–9. IEEE, 2008.

[48] Frederic Stumpf, Andreas Fuchs, Stefan Katzenbeisser, and Claudia Eckert. Improving the scalability of platform attestation. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 1–10. ACM, 2008.

[49] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. Security namespace: making linux security frameworks available to containers. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1423–1439, 2018.

[50] Su Tao. *Trust and integrity in distributed systems*. PhD thesis, Politecnico di Torino, 2017.

[51] PaX Team. Address space layout randomization (aslr). *ht tp://pax. grsecurity. net/docs/aslr. txt*, 2003.

[52] Xin Wan, Zhiting Xiao, and Yi Ren. Building trust into cloud computing using virtualization of tpm. In *Multimedia Information Networking and Security (MINES), 2012 Fourth International Conference on*, pages 59–63. IEEE, 2012.

[53] Miguel G Xavier, Marcelo V Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240. IEEE, 2013.

[54] Mingwei Zhang, Daniel Marino, and Petros Efstathopoulos. Harbormaster: Policy enforcement for containers. In *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*, pages 355–362. IEEE, 2015.