2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering

Building Trust in Container Environment

Yunlong Guo*†, Aimin Yu*, Xiaoli Gong‡, Lixin Zhao*†, Lijun Cai*, Dan Meng*
* *Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China*
† *School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China*
‡ *School of Cyber Security, Nankai University, Tianjin, China*
{*guoyunlong,yuaimin,zhaolixin,cailijun,mengdan*}*@iie.ac.cn, gongxiaoli@nankai.edu.cn*

***Abstract*—Container technology is an emerging virtualization technology that is more efficient and lightweight than virtual machines. This technology is becoming increasingly popular. However, containers are vulnerable to attackers due to various security issues. It is necessary to build trust between users and their containers, as well as between a remote party and the container in the untrusted container environment.**

The existing trusted computing technologies build a trust chain from hardware to the running operating system. In this paper, we extend the trust chain to containers and build trust in the container environment. We first boot the computer to a trusted operating system. The trusted OS then verifies programs running on the OS to improve security. We design the well-formed signature list to help users to authorize container executables and modify the Linux kernel to verify executables running in containers. Therefore, the user is able to control what programs can run in his containers and trust the containers. Our approach generates a measurement list and creates a vTPM for each container. A remote party can request the measurement list based on our container state challenge protocol. As a result, a remote party is able to know the container state and decide whether to trust the container. We also implement the executables measurement and verification mechanism and evaluate the performance. The results show the container start delay is no more than 3% of the normal container start time and the overhead to measure and verify executables is no more than 1µs in most cases, which is reasonably efficient.

***Index Terms*—container, trusted computing, secure boot, measurement and verification, remote attestation**

1. Introduction

Container-based virtualization is an operating system level virtualization technology. Compared with traditional virtual machines on hypervisors, containers running on host operating systems directly have higher performance, better portability and faster speed [1]. In the container environment, the container runtime, such as Docker [2] and rkt [3], creates containers on a host. The orchestration, such as Kubernetes [4] and Docker Swarm [5], schedules and manages containers in the cluster. In recent years, containers are becoming increasingly popular. Gartner predicts by 2020 more than 50% of global organizations will run containerized applications in production [6].

However, containers are more vulnerable due to sharing of the operating system kernel. Attackers may exploit the container runtime vulnerabilities [7] [8] [9] to destroy a container. Besides, other security issues have been found in the container environment [10] [11] [12]. A recent survey shows in 2018, 60% of IT staff working with containers experienced at least one container security incident [13]. Then the question is in an untrusted container environment, how users can trust their containers (e.g. The user wants to confirm that no malware is running in the container.) and how a user or a container can trust other containers (e.g. The user wants to confirm a specific version of the web server is running in a container before uploading secret data.).

Some approaches about how a challenger (a remote party) can trust a system have been studied. The Trusted Computing Group (TCG) [14] defines the coprocessor Trusted Platform Module (TPM) [15] to provide hardware-level security and designs a set of standards to establish trust in a commodity computer. When the system is powered up, it first transfers control to a trusted immutable base, namely Core Root of Trust for Measurement (CRTM). Before transferring control to BIOS, the CRTM measures BIOS code and accumulates the measurement result into the TPM. The process of measuring before transferring control will continue until the operating system controls the system. This trusted boot procedure builds a trust chain from the CRTM to the operating system. Then the challenger can request the measurement stored in the TPM through the secure attestation protocol and judge if the operating system is trusted. Integrity Measurement Architecture (IMA) [16] extends the trust chain to the application layer. IMA measures all executables running on the operating system and uses the TPM to protect the measurement list. With the measurement list, the challenger can know all programs running on the system and decide whether to trust the system. Other architectures [17] [18] [19] have also been proposed to build trust in commodity systems.

In the container environment, there are two main problems with existing methods. First, the remote attestation protocol is not enough in terms of how users can trust their containers. Compared to knowing what programs are

2324-9013/19/$31.00 ©2019 IEEE
DOI 10.1109/TrustCom/BigDataSE.2019.00011

IEEE computer society

running in containers, the users prefer to control what programs can run in their own containers. Second, the current trust chain does not cover containers. It is hard to report the container state to a challenger.

This paper tries to build trust at the container level. First of all, the computer is booted securely to a trusted operating system and only a set of well-examined programs are allowed to run on the system through run-time verification. Before creating containers, the user generates a signature list to authorize programs which can run in specific containers. We modify the Linux kernel to verify container executables against the corresponding signature list. As a result, users can trust their containers for they control what programs can run in these containers. In addition, we measure all executables and generate a measurement list for each container and the host. Based on the virtualized TPM (vTPM), we design the remote attestation protocol for containers to report the container state. At last, we implement the executables measurement and verification mechanism in the Linux kernel and evaluate the performance. The results show our approach is reasonably efficient.

Contributions: in summary, this paper makes the following contributions:

- We design a mechanism that allows users to control what programs can run in their containers, this builds trust between users and their containers
- We design a mechanism that reports the container state to a challenger securely, this extends the traditional trust chain to containers and builds trust between containers and the challenger
- We modify the Linux kernel to implement the executables measurement and verification mechanism and evaluate the performance on the real system

The rest of this paper is organized as follows: Section 2 reviews the related work. Section 3 discusses the threat model. In section 4, we build the public key infrastructure. Section 5 discusses how to secure the host operating system. Section 6 presents the mechanism to measure and verify executables. In section 7, we design the container state attestation protocol. Section 8 implements the executables measurement and verification mechanism and evaluates the performance. Section 9 is the conclusion and future work.

2. Related Work

The related work includes the efforts to prevent unauthorized software from running in containers and build trust on a system.

2.1. Container Programs Security

Clair [20] and Docker Security Scanning [21] scan container images to ensure they are free from known security vulnerabilities or exposures. But malware from the Internet may still run in containers. We need run-time verification. Reference [22] scans vulnerabilities in running containers. This approach can find malware instead of preventing it. SCONE [23] protects container processes from outside attacks with the SGX trusted execution support of Intel CPUs. Nevertheless, it does not verify whether the software in the container is malicious.

2.2. Trusted Computing

The TPM [15] chip is able to measure and authenticate a system. It is always used as the root of trust. The software vTPM [24] virtualizes TPM and can be used in virtualized environments such as virtual machines and containers. AEGIS [25] bootstraps the computer from the trusted code stored in ROM to a trusted operating system kernel. TPod [17] provides a secure software platform by combining the dedicated security hardware, a security operating system kernel and an open security protocol. By using a trusted virtual machine monitor, Terra [18] regards a part of virtual machines as opaque special-purpose platforms and protects the privacy and integrity of their contents. Terra can also report software running in these virtual machines to remote parties. Microsoft's Next Generation Secure Computing Base (NGSCB) [19] partitions the platform into a trusted part and an untrusted part. Programs in the trusted part will be measured. Based on TPM, IMA [16] measures programs running on the operating system and reports the measurement list to the challenger. These research efforts extend the trust chain to the application layer on the operating system.

As for remote attestation, Reference [26] illustrates the high-level TPM-based attestation procedure. Flicker [27] can provide meaningful, fine-grained attestation of the code and its inputs and outputs to a remote party. Reference [28] links specific properties of a remote system to secure tunnel endpoints to improve attestation security. However, these approaches are not designed for containers.

Other efforts try to extend the trust chain to containers. IMA namespacing schemas [29] [30] [31] in the Linux kernel mailing list aim to extend IMA to containers, but they cannot generate the measurement list for each container. Reference [32] discusses how to connect the vTPM to the container. This work does not involve software measurement and the remote attestation protocol. The architecture in Reference [33] creates lightweight containers on demand and extends attestation to runtime properties with tools in the OpenSolaris kernel. However, this architecture only works on OpenSolaris and cannot prevent untrusted programs from running in containers. Tectonic Distributed Trusted Computing [34] tries to establish trust in the entire container cluster, but it is for rkt and cannot report the container state.

3. Threat Model

Our approach focuses on the container environment. In this scenario, multiple containers run on a host operating system kernel. Since the existing trusted computing technologies mentioned in Section II can resist attacks at the host operating system level, we focus on attacks at the container level in this paper. We regard the security components in

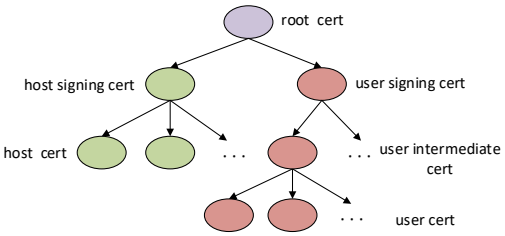

Figure 1. An example of the PKI certificate tree

the kernel proposed in this paper as the Trusted Computing Base (TCB), including the executables measurement and verification component and the vTPM instance. Attackers cannot compromise these components.

Our threat model assumes attackers can compromise containers and run any malicious software in containers. Based on the compromised container, attackers can compromise the host through container escape and run malicious software on the host. In addition, attackers can intercept and modify the network traffic of containers and the host (e.g. remote attestation data). Our threat model does not consider Denial-of-Service attacks, since attackers can simply stop all services running in the container.

4. Build Public Key Infrastructure

Our approach is based on the digital signature algorithm. In this algorithm, each user needs a key pair. The private key is used to generate the digital signature, and the public key is to verify the signature. Our approach leverages the digital certificate (e.g. X.509 certificate) to bind the identity information and the corresponding public key. We also build the public key infrastructure (PKI) [35] to manage digital certificates.

There are two kinds of certificates in the PKI, host certificate and user certificate. The host certificate is held by the administrator of the container platform to authorize programs which can run on the host operating system. The user certificate is held by users to authorize programs that can run in their own containers. Fig. 1 illustrates an example of the certificate tree of the PKI. The root certificate is a self-signed certificate. To protect the root certificate, we generate two signing certificates. The host signing certificate issues host certificates for each host of the container platform. Depending on the application scenario, the user signing certificate may issue user certificates directly or issue user intermediate certificates. There can exist one or more user intermediate certificate layers. The certificate in each layer is issued by the upper layer. This schema can simplify the management of user certificates. Finally, every host and every user will have their own certificate and private key.

The user certificate and corresponding private key should be distributed to users securely. We can distribute certificates through some offline way such as the USB flash disk. This is pretty secure but not practical in some scenarios (e.g. public container clouds). Key graphs [36] are well established for

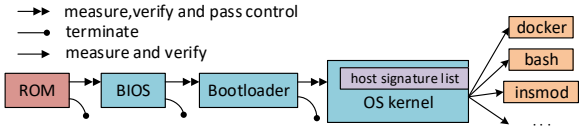

Figure 2. The measurement and verification process, starting from the immutable ROM

(name)	(hash)	(signature)
docker	2ae4b1b539b0540bc60fd20e9dca896344aeb164	ZqkJs97IY0...azmA==
bash	2923d129fd912fc501c0cd1b37172c9091d29b0a	V3xsUSuA+...E+w==
insmod	e3287d7ddddddfb4782fe8a83c39338f3af419a7	WO8Fv2J...wewg==
...		

Figure 3. An example of the host signature list

secret management. This method distributes secrets through the network and can be used in both private and public environments. Moreover, key graphs support certificate update and work well with a large number of users. Therefore, we establish key graphs to distribute certificates.

5. Secure the Operating System

To build trust in the container environment, the first step is to secure the operating system. Fig. 2 shows how to boot a secure operating system. After the machine is powered up, the immutable code stored in ROM is executed first. The code then measures the BIOS and verifies the measurement result against the good values stored in the ROM. If the BIOS passes verification, the code gives the system control to the BIOS. Otherwise, the code will terminate this boot process. The procedure will continue until the OS kernel gains the system control. Any verification failure during the boot process will cause the boot to terminate. As a consequence, the secure boot process ensures only trusted operating systems can run on the machine.

For each host, the administrator generates a host signature list with the host private key. The signature list contains all programs that can run on the host operating system directly (not in containers). Fig. 3 shows an example of the host signature list. Each entry in the list has three parts: program name, program hash and program signature. The program name is designed for the administrator to understand the signature list. It is useless when verifying the program. The program hash is used to identify the program and accelerate the verification process in the OS kernel. It is represented as a hexadecimal sequence. The program signature is utilized to authorize the program. We represent it in a base64 sequence.

The signature list and host certificate are compiled into the kernel so that no malicious users can tamper with them. Through the secure boot discussed above, we only allow OS kernels with the signature list and host certificate to run on the machine. When a software is to run on the operating system and outside the container, the OS kernel

verifies it with the embedded signature list and certificate. The verification process, cache design and other details will be discussed in the next section.

In the container environment, it is possible to allow only a subset of programs to run on the operating system. As the container OS (e.g. RancherOS [37]) does, we only run the necessary software on the operating system. They are required drivers, several kernel modules, system services (e.g. systemd), the container runtime (e.g. Docker), the shell (e.g. bash), some common tools (e.g. ssh), etc. All other business applications should run in the container.

The secure boot and run-time verification mechanism illustrated in Fig. 2 can secure the operating system effectively. The administrator is able to analyse the host signature list to exclude malicious software and vulnerable software. As a result, attackers cannot run malicious software on the operating system and it will be very hard to compromise the software on the host.

6. Measure and Verify Executables

6.1. Authorize Executables

The host executables (not in containers) are authorized based on the host signature list by the administrator, as discussed in Section 5. In this section, we will discuss how to authorize container executables in detail.

We authorize container executables based on container images instead of containers. Because containers are frequently created and destroyed, it is difficult to authorize executables that can run in a particular container. For each container image, the user generates a signature list with the private key which has the same format as the host signature list, namely container signature list. The list contains all programs that can run in containers instantiated from the corresponding container image.

It is necessary to bind the container image and the container signature list. Otherwise, attackers can replace the signature list with another authorized list generated by the same user or add signature entries to the list. Fig. 4 shows the process to bind them. The user packages his certificate into the container image and calculates the fingerprint of the image. The fingerprint may be the image digest or other data that identifies the image. Then, the user calculates a binding signature based on the fingerprint and container signature list with his private key and appends the binding signature to the container signature list. After pushing the container image, the user uploads the container signature list with the binding signature to the container platform for future use.

6.2. Load Certificate and Signatures

As discussed in Section 5, the host certificate and the host signature list are compiled into the kernel. In this section, we focus on how to load the use certificate and the container signature list into the kernel when starting a container.

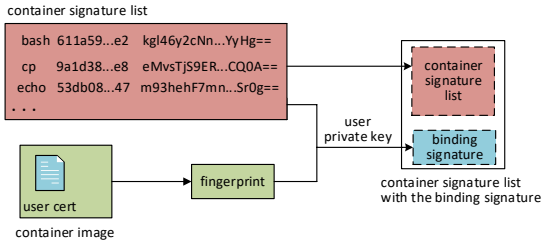

Figure 4. Container image and container signature list binding process

Before loading the user certificate and container signatures into the kernel, we first verify the binding signature. When creating containers, the container runtime extracts the user certificate from the container image and calculates the image fingerprint. The container runtime then uses the signature list, the image fingerprint and the user certificate to verify the binding signature. If the verification fails, the container runtime will refuse to create the container.

We leverage the filesystem securityfs to load the verification data into the kernel. Securityfs is a virtual filesystem in memory mounted at */sys/kernel/security/*. It is designed for kernel security modules. We create the file *ns_verify* in the filesystem. After verifying the binding signature, the container runtime writes the mount namespace id (discussed next), the user certificate and the container signature list into the file *ns_verify*. The kernel will receive these data immediately. The file *ns_verify* is protected with SELinux so that only the container runtime can write data into the file.

6.3. Identify Containers and Host in the Kernel

It is necessary to identify the container in the kernel for measurement and verification. In the Linux kernel, containers are namespaces with limited system resources. In current Linux, there exist six kinds of namespaces: UTS namespace, User namespace, PID namespace, IPC namespace, Network namespace, Mount namespace. But only the Mount namespace is different for different containers. Through the container runtime command (e.g. Docker command), a container can share the other five namespaces with the host or other containers easily. Therefore, we identify containers in the kernel with the mount namespace id.

To measure and verify programs running on the host operating system directly (not in containers), we need to identify these programs. We create the hash table *host_ns_htable* in the kernel. The hash table contains all mount namespace ids that are not created by the container runtime. It is updated with the creation and deletion of Mount namespaces. If the mount namespace id of a program is in the hash table, then it runs on the operating system. For simplicity, we use the pseudo mount namespace id *0x0000* to represent all namespace ids in the hash table.

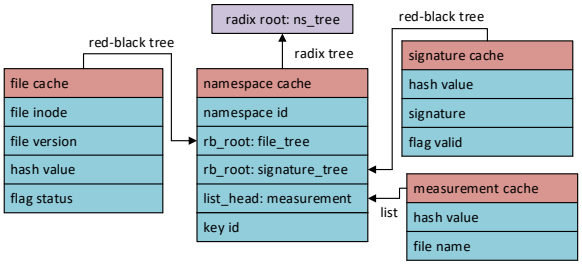

Figure 5. The measurement and verification caches for each namespace

6.4. Organize Data in the Kernel

We employ the Kernel Key Retention Service [38] to manage the host certificate and user certificates in the kernel. In the service, keyring is a special key that contains a list of other keys. We create the keyring *ns_keyring* to link the certificates. At boot time, the host certificate compiled into the kernel is linked to the keyring *ns_keyring*. As for user certificates, we first load the root certificate, the user signing certificate and all user intermediate certificates into the kernel at system startup. When loading user certificates, the keyring *ns_keyring* verifies these leaf certificates with upper certificates in the certificate tree before linking them. Once a certificate is linked, the service generates a key id which can be utilized to get the certificate.

We design four caches to organize other measurement and verification data in the kernel, as shown in Fig. 5. Each container has a namespace cache, which contains the mount namespace id, the key id of the certificate and the roots to find the other three caches. As discussed above, the host operating system also has a namespace cache with the pseudo namespace id *0x0000*. All namespace caches are organized in a radix tree rooted in a global value and the key is the namespace id. Each file to be executed in the container or the host has a file cache, which contains the file inode, the file version, the file hash value and a flag *status*. The file version indicates if the file has been modified to avoid recalculating the file hash. The flag *status* stores the verification result. All file caches in a namespace or pseudo namespace are organized in a red-black tree rooted in the namespace cache and the key is the file inode. Each entry in the signature list has a signature cache, which contains the program hash, the program signature and a flag *valid*. The flag *valid* indicates if the hash value matches the signature to avoid verifying the signature twice. All signature caches in a signature list are organized in a red-black tree rooted in the namespace cache and the key is the hash value. Each program running in the container or the host has a measurement cache, which contains the program hash value and the program name. All measurement caches of a namespace or pseudo namespace are organized in a linked list rooted in the namespace cache.

Algorithm 1 measurement and verification procedure

Input: inode
Output: $PASS$ or $REJECT$

$ns\_id \leftarrow get\_mnt\_ns\_id(current)$
if $in\_host\_htable(ns\_id)$ **then** $ns\_id \leftarrow 0x0000$
end if
$nsc \leftarrow find\_ns\_cache(ns\_id, ns\_tree)$
$filec \leftarrow find\_file\_cache(inode, nsc.file\_tree)$
if $filec \neq NULL$ **and** $eq\_version(inode, filec)$ **then return** $filec.status$
end if
if $filec = NULL$ **then**
    $filec \leftarrow insert\_file\_cache(nsc.file\_tree)$
end if
$hash \leftarrow cal\_file\_hash(inode)$
$update\_file\_cache(filec, inode, hash)$
$sigc \leftarrow find\_sig\_cache(hash, nsc.signature\_tree)$
if $sigc = NULL$ **then**
    $filec.status \leftarrow REJECT$ **return** $REJECT$
end if
if $sigc.valid = UNKNOWN$ **then**
    $key \leftarrow get\_key(nsc.key\_id)$
    $valid \leftarrow verify(sigc.hash, sigc.signature, key)$
    $sigc.valid \leftarrow valid$
end if
if $sigc.valid = MATCH$ **then**
    $filec.status \leftarrow PASS$
    $append\_entry(hash, inode, nsc.measurement)$
    if $ns\_id = 0x0000$ **then**
        $extend\_TPM(pcr, hash)$
    else
        $extend\_vTPM(pcr, hash, ns\_id)$
    end if
    return $PASS$
else
    $filec.status \leftarrow REJECT$ **return** $REJECT$
end if

6.5. Measurement and Verification Process

We measure and verify executables in the Linux Security Modules (LSM) hooks. The LSM hook *mmap_file* is responsible for binary programs and dynamic link libraries and the LSM hook *bprm_check_security* is for executable scripts. Alg. 1 shows the measurement and verification procedure. The procedure takes the file inode as input, measures and verifies the file, then outputs the verification result. The procedure first checks whether the file runs on the operating system and sets current mount namespace id to the pseudo mount namespace id if so. If the file has been measured and verified before and not modified, the procedure does not extend the measurement list and returns the verification result in the flag *status*. Otherwise, it measures the file and finds the signature cache based on the file hash value. If no signature cache is found or the signature is invalid, the procedure will return REJECT directly. But if the signature is valid, the procedure will append the measurement to the

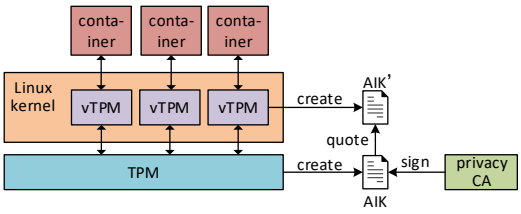

Figure 6. The connection between vTPM instances and containers

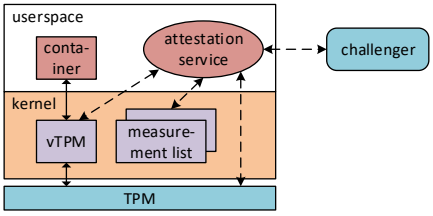

Figure 7. The architecture for container state attestation

measurement list and extend the corresponding TPM or vTPM before returning PASS.

6.6. Security Analysis

With the host signature list, the administrator only allows the container runtime that verifies the binding signature and loads the verification data to run on the host. Attackers cannot run a tampered container runtime. The user binds the container signature list and the container image through the binding signature so that no attackers can tamper with the program signatures. When loading user certificates, the kernel verifies if the certificate is legal to avoid loading untrusted certificates. As a result, the user is able to control what programs can run in his containers exactly.

7. Container State Attestation

The procedure in Alg. 1 generates a measurement list for each container and the host. This list contains all programs that have run in the container or host. In this section, we discuss how to report the container measurement list and host measurement list to a remote party securely.

7.1. Connect vTPM to the Container

There exist two architectures on how to connect vTPM instances and containers [32]. One is to run vTPM instances in user space. In this architecture, each container runs a driver client. The driver client communicates with a vTPM manager that is placed on the host or in a special container to manage the vTPM instance. The other is to run vTPM instances in the kernel. In this architecture, applications in the container are able to communicate with the vTPM instance directly. In our scenario, users control programs which can run in their containers. Hence it is difficult to run the driver client in each container. Moreover, it is not secure enough to run vTPM in user space. Considering feasibility and security, we create a vTPM instance for each container in the kernel, as illustrated in Fig. 6.

The vTPM should create a valid Attestation Identity Key (AIK) for remote attestation. For the hardware TPM chip, it creates the AIK pair and sends the public part to the privacy certificate authority (privacyCA). The privacyCA then issues an AIK certificate which can be used to identify the TPM. But in the container environment, containers and vTPM instances are created and destroyed frequently. It is not so efficient to request the AIK certificate from the remote privacyCA. We use the second solution proposed in Reference [24]. As shown in Fig. 6, the vTPM first creates an AIK'. Then it sends the TPM_Quote request to the TPM to validate the AIK'. This quote binds the AIK' and the identity of the container that connects the vTPM. A remote party can verify the AIK' of vTPM with the valid AIK owned by TPM.

7.2. Container State Challenge Protocol

We aim to extend the trust chain from the host OS to the container. To trust a container, the challenger needs to know all the measurement results on the trust chain, including BIOS, BootLoader, OS kernel, programs running on the OS and programs running in the container. The challenger can request the appropriate TPM PCRs to obtain the measurement results of BIOS, BootLoader or the OS kernel according to the traditional challenge process. In this section, we report the host measurement list and container measurement list to the challenger.

The architecture for container state attestation is shown in Fig. 7. We deploy an attestation service on the host operating system rather than in the container, since users may not allow the service to run in their containers and it is a bit bulky to run an attestation service in each container. The service is able to communicate with the TPM and all vTPM instances. It can also access the host measurement list and all container measurement lists in the kernel. The attestation service is responsible to report the container state to the challenger.

The container state challenge protocol is shown in Fig. 8. In step 1, the challenger generates a non-predictable 160bit random *nonce*. In step 2, the challenger sends the challenge request, which contains the random *nonce* and the identity of the target container, to the attestation service. The AS prepares attestation data in step 3. It first loads the protected AIK_{priv} into TPM. Then the AS requests the *HQuote* from the TPM which signs the host *PCR* and the *nonce* provided by the challenger. Next it requests the host measurement list in the kernel. With the same process, the AS obtains the *CQuote* from the corresponding vTPM and the container measurement list from the kernel. In step 4, the AS sends the challenge response to the challenger, which contains *HQuote*, *HML*, *CQuote*, *CML*, $Quote_{AIK'}$ that is requested by the vTPM from the TPM and AIK'_{pub} that is the public

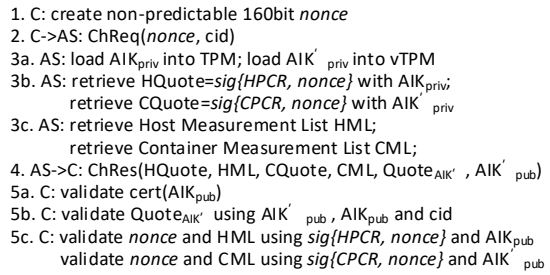

Figure 8. Container state challenge protocol

part of the vTPM AIK. In step 5, the challenger validates the attestation data. First, the challenger validates the certificate $cert(AIK_{pub})$ and compares the identification in this certificate with the identification of AS. This makes sure that the challenger is communicating with the right system. Then the challenger validates the $Quote_{AIK'}$ with AIK'_{pub}, the valid AIK_{pub} and the target container identity. This ensures the *CQuote* and container measurement list belong to the target container. To validate the *nonce* and host measurement list, the challenger recalculates the TPM aggregate based on the host measurement list. Then the challenger verifies $sig\{HPCR, nonce\}$ with the *nonce*, TPM aggregate calculated before and AIK_{pub}. If the verification succeeds, the host measurement list is valid and not tampered with, otherwise it is invalid. The challenger validates the container measurement list in the same way. After getting the valid host measurement list and container measurement list, the challenger can decide whether to trust the container based on programs on these measurement lists.

7.3. Security Analysis

The container state challenge protocol can resist multiple attacks. For each challenge request, the challenger creates a non-predictable random nonce. Therefore, attackers cannot return previous attestation data (*replay attack*). The validation of the AIK and AIK' can prevent attackers from replacing the original attestation data with the data of another system (*masquerading*). By verifying the signature in HQuote and CQuote, the protocol is able to discover the illegal modification of the measurement list and TPM aggregate (*tampering*). The protocol can also prevent *reboot attacks* to some extent. Since the container does not know when the attestation happened, it is difficult to restart the container just after the attestation data are returned.

8. Implementation and Performance

8.1. Implementation

We implement the executables measurement and verification mechanism on the platform with CentOS 7.2 and Docker 17.06. Runc [39] is the software layer between the container manager (e.g. Docker) and the operating system kernel. We modify runc to verify the binding signature and load verification data into the kernel. In our implementation, the fingerprint is the container image specification that contains image layer hash values and image configurations. We add two *pivot_root* calls in runc to obtain the mount namespace id in the container rootfs and load verification data in the host rootfs. The modification of runc consists of 262 lines of Golang code including comments. We modify the Linux kernel to enforce the measurement and verification based on the existing IMA implementation. The IMA process is changed to execute our functions that measure and verify executables. The implementation consists of 1710 lines of C code including comments. Our modification is based on the Linux kernel 4.17.14, but it is easy to port to other Linux kernel versions.

Table 1. CONTAINER START DELAY

Image	Reference (stdev)	Binding Delay (stdev)	Loading Delay (stdev)	Signature Entry Number	Total Delay
hello-world: latest	268.9ms (5.50)	3.59ms (0.21)	0.25ms (0.02)	1	3.84ms
nginx: 1.13	270.6ms (5.64)	5.20ms (0.15)	2.13ms (0.05)	747	7.33ms
ubuntu: 16.04	274.4ms (7.68)	5.64ms (0.19)	2.44ms (0.06)	894	8.08ms

8.2. Performance Evaluation

Reference [16] has examined the overhead to extend the TPM aggregate. In this section, we evaluate the container start delay and executables measurement and verification overhead without extending the TPM or vTPM. All experiments are carried out on an Inspur server, including 16 Intel Xeon 2.1 GHz processors and 64 GBytes of RAM.

8.2.1. Container Start Delay. The container start time here is the time from scratch to the first process in the container runs successfully, which includes the creation time and start time in Docker context. Table 1 shows the experimental results. The reference is the normal Docker container start time with the original runc. It is measured with the *docker run -d* command. The binding delay is the time to verify the binding signature. The loading delay is the time to load the signature list and certificate into the kernel. We measure them by inserting measurement functions in runc. The signature entry number is the number of signature entries in the corresponding signature list. The signature list used in our experiment contains all executables in the corresponding container image. The total delay is the sum of the binding delay and loading delay.

Since we use the image specification managed by the Docker engine as the fingerprint rather than calculating the image hash value, it takes less than 6ms to verify the binding signature. As the signature entry number increases, both the binding delay and loading delay (as a result, the total delay) increase. Nevertheless, they grow very slowly. For

Table 2. EXECUTABLES MEASUREMENT AND VERIFICATION OVERHEAD

File Size (Bytes)	measure_verify (stdev)	use_valid (stdev)	use_status (stdev)
2	264.7µs (4.1)	15.9µs (0.2)	0.88µs (0.0)
512	265.4µs (1.5)	16.2µs (0.5)	0.95µs (0.0)
1K	265.8µs (1.5)	17.1µs (0.2)	0.95µs (0.0)
16K	291.3µs (0.8)	45.1µs (0.6)	1.0µs (0.0)
128K	500.6µs (1.5)	252.2µs (0.8)	1.0µs (0.0)
1M	2170.2µs (3.3)	1953.4µs (3.3)	1.0µs (0.0)

example, loading 747 signature entries and the certificate into the kernel takes 2.13ms and loading 894 entries only takes 2.44ms. From the table, we can conclude that the total delay is no more than 3% of the normal container start time, which is pretty efficient.

8.2.2. Executables Measurement and Verification Overhead. We insert measurement functions in the Linux kernel and use ubuntu:16.04 to evaluate the overhead to measure and verify executables without extending the TPM or vTPM aggregate. Table 2 shows the results. Our overhead measurement considers three cases: measure and verify the file, measure the file and use the signature verification result cached in flag *valid*, use the file verification result cached in flag *status*. We need to append the file measurement result to the measurement list in the first two cases. In Table 2, measure_verify represents the first case, this happens when a new file or a modified file first runs and the signature verification result is unknown. We use use_valid to represent the second case, it happens when a new file or a modified file first runs but the corresponding signature has been verified. The third case is represented as use_status, this happens when a file measured and verified before runs again.

The time to measure a file depends on the file size. Therefore, when the file size is large, the overhead of measure_verify and use_valid may be significant. For example, the measure_verify overhead is 2170.2µs and the use_valid is 1953.4µs when the file has 1 Megabyte. Fortunately, these two cases account for less than 0.1% of all measurement and verification calls [16]. The time to verify a signature is about 250 µs (measure_verify - use_valid) regardless of the file size. Since the case use_status does not measure the file or verify the signature, the use_status overhead is independent of the file size and no more than 1µs, which is pretty minimal.

9. Conclusion and Future Work

In this paper, we propose an approach to build trust in the container environment. The approach regards the immutable code in ROM as the root of trust and only allows the trusted operating system to run on the machine. Based on the well-formed signature list, the administrator can authorize programs for the host operating system and users can authorize programs for their containers. We design caches and modify the Linux kernel to verify executables securely and efficiently. For each container, the approach generates a measurement list and instantiates a vTPM. A remote party can request the container measurement list and host measurement list securely through the container state challenge protocol. We implement the executables measurement and verification mechanism in the kernel and evaluate the performance. The results show the overhead is reasonably efficient. As a conclusion, users are able to control what software can run in their containers and trust the containers. A remote party is able to know the container state and decide whether to trust the container according to his own criteria.

Currently, the administrator controls programs that can run on the host operating system. However, users may not trust these programs. We plan to allow users to generate an untrusted program list for each container image. The orchestration then manages to schedule containers instantiated from the container image to the host that does not run programs on the untrusted program list.

Acknowledgments

This work is supported by the National Key R&D Program of China (2016YFB0801001).

References

[1] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. IEEE, 2015, pp. 171–172.

[2] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[3] "rkt Overview." *rkt Documentation*, [Online]. Available: https://coreos.com/rkt/

[4] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Queue*, vol. 14, no. 1, p. 10, 2016.

[5] "Docker Swarm." *Docker Documentation*, [Online]. Available: https://docs.docker.com/swarm/

[6] "6 Best Practices for Creating a Container Platform Strategy." *Gartnet Blog*, [Online]. Available: https://www.gartner.com/smarterwithgartner/6-best-practices-for-creating-a-container-platform-strategy/

[7] "CVE-ID: CVE-2016-3697." *MITRE Website*, [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3697

[8] "CVE-ID: CVE-2016-9962." *MITRE Website*, [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9962

[9] "CVE-ID: CVE-2019-5736." *MITRE Website*, [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5736

[10] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on docker hub," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy,* 2017, pp. 269–280.

[11] J. Gummaraju, T. Desikan, and Y. Turner, "Over 30% of official images in docker hub contain high priority security vulnerabilities," in *Technical Report,* BanyanOps, 2015.

[12] A. R. Manu, J. K. Patel, S. Akhtar, V. K. Agrawal, and K. B. S. Murthy, "A study, analysis and deep dive on cloud PAAS security in terms of Docker container security," in *Circuit, Power and Computing Technologies (ICCPCT), 2016 International Conference on,* 2016, pp. 1–13.

[13] "Tripwire State of Container Security Report." *Tripwire Website*, [Online]. Available: https://www.tripwire.com/solutions/devops/tripwire-dimensional-research-state-of-container-security-report/

[14] "Trusted Computing Group." *TCG Website*, [Online]. Available: https://trustedcomputinggroup.org/

[15] "TPM Main Specification." *TCG Website*, [Online]. Available: https://trustedcomputinggroup.org/resource/tpm-main-specification/

[16] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn, "Design and Implementation of a TCG-based Integrity Measurement Architecture." in *USENIX Security symposium*, vol. 13, 2004, pp. 223–238.

[17] H. Maruyama, F. Seliger, N. Nagaratnam, et al. "Trusted platform on demand (TPod)". IBM, 2004.

[18] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *ACM SIGOPS Operating Systems Review,* 2003, vol. 37, pp. 193-206.

[19] P. England, B. Lampson, J.Manferdelli, and B. Willman, "A trusted open platform" in *Computer,* vol. 36, no. 7, pp. 55-62, 2003.

[20] "Clair: Automatic container vulnerability and security scanning for appc and Docker," *Clair Documentation*, [Online]. Available: https://coreos.com/clair/docs/latest/

[21] "Docker Security Scanning," *Docker Documentation*, [Online]. Available: https://docs.docker.com/docker-cloud/builds/image-scan/

[22] N. Bila, P. Dettori, A. Kanso, Y. Watanabe, and A. Youssef, "Leveraging the serverless architecture for securing linux containers," in *Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on,* 2017, pp. 401-404.

[23] S. Arnautov et al, "SCONE: Secure Linux Containers with Intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16),* 2016, pp. 689-703.

[24] R. Perez, R. Sailer, and L. van Doorn, "vTPM: virtualizing the trusted platform module," in *Proc. 15th Conf. on USENIX Security Symposium*, 2006, pp. 305–320.

[25] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*. IEEE, 1997, pp. 65–71.

[26] B. Parno, J. M. McCune, and A. Perrig, "Bootstrapping trust in commodity computers," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 414–429.

[27] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *ACM SIGOPS Operating Systems Review,* 2008, vol. 42, pp. 315-328.

[28] K. Goldman, R. Perez, and R. Sailer, "Linking remote attestation to secure tunnel endpoints," in *Proceedings of the first ACM workshop on Scalable trusted computing - STC 06,* Alexandria, Virginia, USA, 2006, p. 21.

[29] Guilherme Magalhaes, "ima: namespace support for IMA policy," *Linux Kernel Mailing List*, [Online]. Available: https://lkml.org/lkml/2017/5/11/699

[30] Mehmet Kayaalp, "ima: namespacing IMA audit messages." *Linux Kernel Mailing List*, [Online]. Available: https://lkml.org/lkml/2017/7/20/905

[31] Stefan Berger, "ima: Namespacing IMA." *Linux Kernel Mailing List*, [Online]. Available: https://lkml.org/lkml/2018/5/11/383

[32] S. Hosseinzadeh, S. Laurn, and V. Leppnen, "Security in container-based virtualization through vTPM," in *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC),* 2016, pp. 214-219.

[33] K. Bailey and S. Smith, "Trusted virtual containers on demand," in *Proceedings of the fifth ACM workshop on Scalable trusted computing,* 2010, pp. 63-72.

[34] J. Wood, "Technical Brief Distributed Trusted Computing," *Tectonic Documentation*, Available: https://tectonic.com/assets/pdf/TectonicTrustedComputing.pdf

[35] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, W. Polk, "Internet X. 509 public key infrastructure certificate and certificate revocation list (CRL) profile." No. RFC 5280. 2008.

[36] C. Wong, M. Gouda, and S. Lam, "Secure group communications using key graphs," in *IEEEACM Trans. Netw.,* vol. 8, no. 1, pp. 16-30, 2000.

[37] "RancherOS: Fast, ultra-lightweight container OS," *Rancher Documentation*, [Online]. Available: https://rancher.com/rancher-os/

[38] "Kernel Key Retention Service." *Linux kernel Documentation*, [Online]. Available: https://www.kernel.org/doc/html/v4.13/security/keys/core.html

[39] "runc: CLI tool for spawning and running containers according to the OCI specification," *runc Documentation*, [Online]. Available: https://github.com/opencontainers/runc