POLITECNICO DI TORINO

Master degree course in Ingegneria Informatica

Master Degree Thesis

Container Attestation with Linux IMA namespaces

Supervisor
Prof. Antonio Lioy
Ing. Silvia Sisinni
Ing. Enrico Bravi

Candidate
Lorenzo Ferro

October 2023

To my family

Summary

In contemporary times, containers play a crucial role in various fields, including cloud computing and microservices, among others. Their popularity continues to grow due to their flexibility, simplified deployment, compatibility with multiple operating systems, rapid availability, and precise allocation of computational resources in microservices. Ensuring the integrity and proper configuration of software on containers is vital for early detection of tampering and breaches, allowing for prompt response to attacks. Remote Attestation is a process through which an external entity evaluates the trustworthiness of a computational node. While effective for physical nodes, it's not yet well-established for virtual nodes, such as containers. Some proposals have been made to address this issue, but they face challenges, such as the inability to verify closed containers, scalability, and performance concerns. The current direction in Linux development is to create a new namespace for the Integrity Measurement Architecture. This thesis aims to leverage this choice by proposing a solution for conducting container attestation based on the IMA namespace. This proposed solution entails modifications to the IMA mechanism to enable remote attestation of a container without divulging information to external parties independently from its status. To ensure the integrity of the containers' list and prevent the establishment of a namespace for unmonitored program execution each time an event occurs, a record of the event is maintained within its parent chain. This includes the namespace that initiated the target, the one that created the parent, and so on, all the way up to the host. This process is conducted without storing specific details about the nature of the event in a manner that allows for continuous traceability while maintaining containers' privacy. The Keylime framework has been adjusted to support this modified version of IMA for attestation. Tests have demonstrated the low latency of the measurement and attestation mechanism, regardless of the number of containers running on the machine. Importantly, this solution is independent of containerization technology.

Acknowledgements

Thanks.

Contents

Chapter 1

Introduction

In contemporary times, containers play a crucial role in various fields, including cloud computing and microservices, among others. Containerization has quickly become a technology that provides organizations with a secure environment for developing, testing, and running applications. An application must work the same as intended in development, test, and production environments to be successful. This can be quite challenging due to each environment having its unique configurations and supporting files. Containers provide a solution by encapsulating all essential information within a single container image, allowing it to run in isolated user environments. Containers and virtual machines differ primarily in their approach to virtualization. Virtual machines achieve virtualization by replicating an entire machine down to the hardware layers using a hypervisor, whereas containers only virtualize software layers above the operating system level.

A growing number of organizations are understanding the importance of interconnected systems. Additionally, adopting these systems is becoming more straightforward as the cost of sensors continues to decrease, and the capabilities of edge and cloud computing steadily improve. Enterprise Internet of Things solutions are swiftly maturing, expanding in both size and scope. However, organizations are encountering distinct challenges concerning the continuity, connectivity, and cybersecurity of IoT devices. Containers' portability, immutability, and isolation of environments make them an optimal solution for addressing these concerns. In addition to simplifying software development for enterprises, containers also handle updates for endpoint devices. Furthermore, containers facilitate the scaling of the IoT environment through a microservices model. However, there are lingering challenges in container security. One of these is ensuring the integrity and correct configuration of software within containers. This is crucial for promptly detecting tampering and breaches, enabling a swift response to potential attacks.

One method for identifying tampering and breaches in standard systems involves employing Remote Attestation [1]. This entails an external entity assessing the reliability of a computational node. This procedure hinges on the Attester, a component of the infrastructure, possessing specific knowledge about the anticipated behavior. The Attester is tasked with transmitting its present state, usually comprising a collection of measurements of the software components active on the system, to another entity known as the Verifier. It is the responsibility of the Verifier to compare this data with the information it already holds, in order to ascertain whether the Attester is in a trusted state or not. This approach can be reinforced by employing secure hardware installed on the device, such as the Trusted Platform Module (TPM) developed by the Trusted Computing Group (TCG). Although effective when applied to physical nodes, Remote Attestation is not yet widely established for virtual nodes, such as containers.

The Linux community has created the Integrity Measurement Architecture (IMA) module, enabling the extension of the "Secure Boot" concept to application execution. The Linux IMA subsystem incorporates hooks into the Linux kernel to facilitate the generation and gathering of file hashes when they are opened, before accessing their contents for reading or execution. IMA keeps track of a dynamic list of measurements during runtime. If it is anchored in a hardware TPM, it also maintains an aggregate integrity value based on this list. By anchoring the aggregate integrity value in the TPM, it becomes notably challenging to compromise the measurement list

through a software attack without detection. Initiatives like Keylime offer a remarkably scalable approach to measured boot attestation and real-time integrity measurement. They also enable Remote Attestation of the IMA measurement list.

Some approaches have been proposed and discussed upon achieving Remote Attestation of containers. One such solution, known as "Container-IMA", enables the independent assessment of the integrity status of individual containers operating on the platform. Nevertheless, it does have its limitations. This solution relies on container PCRs, with each container possessing its PCR. This proves especially effective in safeguarding the confidentiality of container measurements within multi-tenant environments, as a Verifier only obtains the measurements about the specific container it needs to attest. However, it lacks the capability to verify closed containers and encounters difficulties when a container is stopped and restarted. Additionally, the privacy assurance of container-related measurements relies on a shared secret between the kernel space of the system hosting the container and the Verifier. The method for securely sharing this secret is not clearly defined.

The ongoing focus in Linux development is on establishing a distinct namespace for IMA. Linux namespaces serve as the foundational technology for container platforms like Docker. They constitute a feature within the Linux kernel that enables the system to confine the resources visible to containerized processes, ensuring they operate independently without interference. The IMA namespace development leaves still some open questions. The present implementation establishes a measurement list for every IMA namespace and allows the customization of policies. However, at this point, there is no implementation in place for the "IMA measure rule".

The objective of this thesis is to harness the capabilities of the Linux IMA namespace by presenting various approaches to perform container attestation using the IMA namespace. Those proposed solutions involve altering the IMA mechanism to facilitate the separate recording of events within the namespace's list. For security reasons, it is imperative to log the measurements taken within the containers in the host's list. Failure to do so could result in a scenario where a container establishes an IMA namespace, executes code within it, and subsequently closes it, rendering it impossible to conduct verifications. The challenging part lies in maintaining a record of these events in the host's list without divulging excessive information about the executed data, particularly to other containers during verification. This must be accomplished without introducing time-consuming operations, as event measurement must be a swift process. Moreover, the thesis also focused on extending Keylime, a remote attestation framework capable of attesting physical nodes through the use of the TPM 2.0 specification. The Keylime code was appropriately adapted to enable the Keylime Verifier to understand the new IMA templates and facilitate the attestation of individual containers by inserting the verification logic necessary for the proposed solutions.

The system underwent comprehensive testing, encompassing both functional and performance assessments. The outcomes indicate that the system operates following expectations, with the additional time overhead to the Keylime framework being negligible. Furthermore, the testing demonstrated that the measurement and attestation mechanism exhibits low latency, and this efficiency scales appropriately with the increasing number of containers active on the machine. Importantly, this solution is independent of containerization technology.

Chapter 2

Trusted Computing and TPM 2.0

Trusted Computing [2] refers to a concept and set of technologies aimed at enhancing the security and trustworthiness of computing systems involving hardware and software mechanisms. Trusted Computing focuses on constructing secure and reliable computing platforms capable of resisting a range of threats and attacks.

Trusted computing pertains to a computer system where there is a certain level of confidence that the system, either in part or entirely, is operating as expected. This assurance can be extended to various entities, such as the human user of the PC or a program executing on a remote machine. The extent of this assurance, whether it encompasses the entire system or only specific components, and the type of entity receiving the assurance, vary depending on the specific system and its operational environment.

The traditional approach to computer security relies on software-based measures such as firewalls, antivirus software, and encryption. While these methods are effective to a certain extent, they are not foolproof and can be vulnerable to sophisticated attacks. Trusted Computing takes a different approach by combining both hardware and software mechanisms to establish and verify the integrity of a computing system.

The application of Trusted Computing is broad and encompasses various fields, including secure remote access, secure cloud computing, digital rights management, secure supply chains, and more. By establishing a chain of trust and implementing robust security measures, Trusted Computing helps mitigate the risks associated with unauthorized access, tampering, and other malicious activities.

2.1 Trusted Computing Base

The term Trusted Computing Base (TCB) [3] encompasses the crucial system components required to establish and maintain security within a specific system. The components within the TCB are labeled as a"Base" due to their role as the fundamental building blocks for the security of the system. They form the foundation upon which the system's security is established and maintained.

Interestingly, a component labeled as "trusted" in a system does not necessarily guarantee security or trustworthiness. In everyday language, the term "trusted" typically implies reliability, truthfulness, or being worthy of confidence. However, in the realm of computer security, "trusted" solely signifies the component's critical role in maintaining security within the system's scope. It is important to note that being labeled as "trusted" does not imply the ability of the component to withstand attacks. Rather, it signifies the component's significant contribution to the overall security of the system. Trustworthiness is primarily established through formal verification or cryptographic measures. These methods utilize mathematical techniques to provide evidence that a system or its components are behaving as intended and can be relied upon.

- **Formal Verification** [4]: employs mathematical techniques to ensure that a design adheres to a precisely defined concept of functional correctness. This approach offers an alternative

method to identify input sequences that violate specified properties and also to prove that these properties consistently hold true when no such stimuli exist. Formal verification can be applied to designs described at various levels of abstraction. There are several use models for formal verification, with some design teams utilizing them to enhance the coverage of random simulation. State-of-the-art tools seamlessly integrate formal verification engines with simulation processes. Ongoing research in formal verification aims to enhance the capacity of design analysis algorithms and facilitate their use earlier in the design cycle.

- **Cryptographic measures**: employing cryptographic functions to verify system integrity during runtime and guaranteeing it remains unaltered. Cryptographic tasks are performed by a secure cryptoprocessor which refers to a specialized computer-on-a-chip or microprocessor. It is integrated into a protective package with various physical security features, providing a certain level of resistance against tampering. Unlike regular cryptographic processors that release decrypted data into a secure environment, a secure cryptoprocessor ensures that decrypted data and program instructions are not exposed in environments where maintaining absolute security may be challenging.

Tamper evidence is another crucial aspect of showcasing trustworthiness. The level of difficulty in detecting tampering becomes higher as one moves toward the top layers of the system stack, away from the hardware. To enhance security, many systems integrate a hardware root of trust, such as a Trusted Platform Module (TPM), which provides a solid foundation for establishing trust and protecting against tampering.

An **attack surface** [5] refers to the collective vulnerabilities, pathways, or methods, commonly known as attack vectors, that can be exploited by hackers to achieve unauthorized access to the network, compromise sensitive data, or execute a cyberattack. It encompasses the various entry points and weak spots that pose potential risks to the organization's security. Considering the concept of attack surface, securing, auditing, or fully assessing a large and intricate TCB becomes more challenging. An optimal TCB is characterized by its small and simple design, while still capable of delivering the required security assurances for the system. This approach ensures easier management and enhances the ability to effectively safeguard the system against potential threats.

2.1.1 Main components

- **Reference Monitor**: a mediator between the system's resources and the processes or entities that seek access to those resources. The primary objective of the Reference Monitor is to ensure that every access to a resource is authorized and consistent with the established security policy. Implemented by the Security Kernel.
- **Security Perimeter**: refers to the defined boundaries within which the TCB operates and enforces security policies. It delineates the extent of the trusted computing environment and determines the scope of protection provided by the TCB.
- **Trusted Paths**: refers to secure communication channels established between users or entities and the TCB. These paths ensure the confidentiality and integrity of sensitive information exchanged during interactions with the TCB.

2.2 Key Concepts

Those are the main concepts that are required for a fully Trusted system, a system that satisfies the TCG (Trusted Computing Group) specifications

- **Memory curtaining**: is a security methodology employed to limit entry to particular segments of computer memory. Its primary objective is to prevent unauthorized access, alteration, or disclosure of confidential data stored in memory. This technique involves partitioning the memory space into distinct and isolated regions or compartments, with each region being allocated precise access permissions.

- **Sealed storage**: is a security feature that enables the encryption and protection of sensitive data stored on a system. It allows to include in the cryptographic operation the state of the platform, ensuring that it remains confidential and can only be accessed when the TPM is in a specific desired configuration. If there are any modifications or discrepancies in the TPM's state, the sealed data remains inaccessible. Commonly used to protect sensitive information, such as cryptographic keys, credentials, or critical system data.
- **Trusted Third Party (TTP)**: is an entity or organization entrusted with the facilitation and supervision of transactions between multiple parties. The primary role of a TTP is to act as a neutral and trusted intermediary, ensuring fairness, security, and trust in various types of transactions.
- **Remote Attestation**: enables authorized parties to detect any changes in a system. It involves generating a hardware-generated certificate that represents the current state of the running software. This certificate can be provided to a remote party as evidence that the software running on the system remains unaltered.
- **Endorsement key**: a pair of 2048-bit RSA public and private keys that are randomly generated during the manufacturing process of the chip. Used for secure transactions, serves as a root of trust, and is used for various security functions within the TPM. It is used to sign attestation data generated by the TPM, providing proof of the system's measured state and configuration. The EK is unique to each TPM and cannot be modified or replaced.
- **Secure Input Output**: is a mechanism used to ensure the confidentiality and integrity of data exchanged between the TPM and other system components during input and output operations. It aims to protect sensitive information from unauthorized access, tampering, or interception.
- **Secure Boot**: is a mechanism that ensures only trusted and digitally signed software is loaded during the system startup process. It verifies the integrity and authenticity of each component, starting from the bootloader, the operating system kernel, and up to the applications. By preventing the execution of unauthorized or modified software, Secure Boot helps protect against "bootkits" and other malware that attempt to compromise the system at an early stage.
- **Root of Trust**: encompass hardware, firmware, and software components renowned for their exceptional reliability in executing vital security tasks. Since roots of trust inherently possess trustworthiness, they necessitate meticulous design to ensure their security. Consequently, numerous roots of trust are implemented in hardware to prevent malicious interference with their essential functions. These roots of trust establish a solid groundwork upon which security and trust can be effectively established.

2.3 Trusted Computing Group TGC

The Trusted Computing Group (TCG) [6] is an industry consortium comprising technology companies and organizations. Its main objective is to create and advance open standards for trusted computing and security technologies. Through collaboration, the TCG establishes specifications, protocols, and best practices to enhance the security and reliability of computing platforms.

The TCG specializes in several aspects of trusted computing, such as hardware-based security components like Trusted Platform Modules (TPMs), secure boot processes, secure storage, remote attestation, and cryptographic operations. Through the establishment of shared standards, the TCG strives to promote interoperability among products from different vendors and facilitate the widespread adoption of trusted computing technologies across diverse platforms and environments. Notable examples of TCG specifications include those related to TPMs.

The TCG's work is influential in shaping the landscape of trusted computing and security technologies. Its standards and specifications are widely adopted by industry stakeholders, including hardware manufacturers, software developers, and security solution providers, to ensure interoperability and improve the security of computing systems and networks.

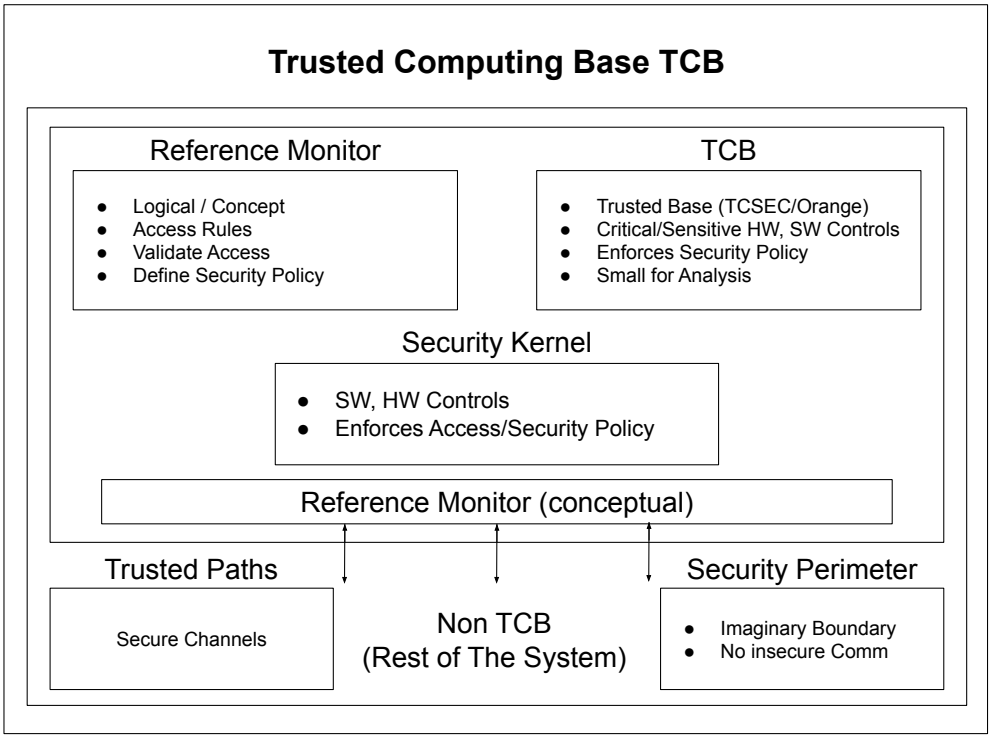

Figure 2.1. TCB summary schema [3]

2.4 TPM 2.0

The Trusted Platform Module (TPM) [7] is a globally recognized standard for a secure cryptoprocessor. It is a dedicated computer-on-a-chip that incorporates multiple physical security measures to provide a certain level of tamper resistance. It is a dedicated microcontroller that safeguards hardware by utilizing integrated cryptographic keys.

2.4.1 Main features

- **Hardware random number generator**: is a device or circuit that produces random numbers using physical processes or phenomena. It is particularly well-suited for applications that demand high-quality random numbers. In contrast to pseudo-random number generators (PRNGs), which depend on deterministic algorithms and seed values, HRNGs generate true randomness by harnessing inherent unpredictability.

- **Generation of cryptographic keys**: keys for limited use, it utilizes random numbers produced by the HRNG alongside specific algorithms. To bolster key security, it may utilize key wrapping methods, which involve encrypting the cryptographic keys generated using an additional key.

- **Remote attestation** [8]: extends the trust of the trusted computing platform beyond the terminal to encompass the entire network. This practice fulfills various security requirements for users, including determining the trustworthiness of the communicating computing platform.

- **Binding**: encryption using the TPM bind key (descended from the storage key), cannot be decrypted outside the TPM. This solution offers strong security since even if the data is compromised, decryption is not possible. However, it also poses challenges when data needs to be exported and transferred to another machine, as it requires a complex procedure.

- **Sealing**: similar to binding but includes also the TPM state as part of the encrypting operation. For decrypting the TPM state has to be the same as the encryption time. The state is the collection of all applications running plus the configuration files.
- **Authenticate hardware devices**: using the `Endorsment key` that is unique to each TPM is possible to identify the machine.

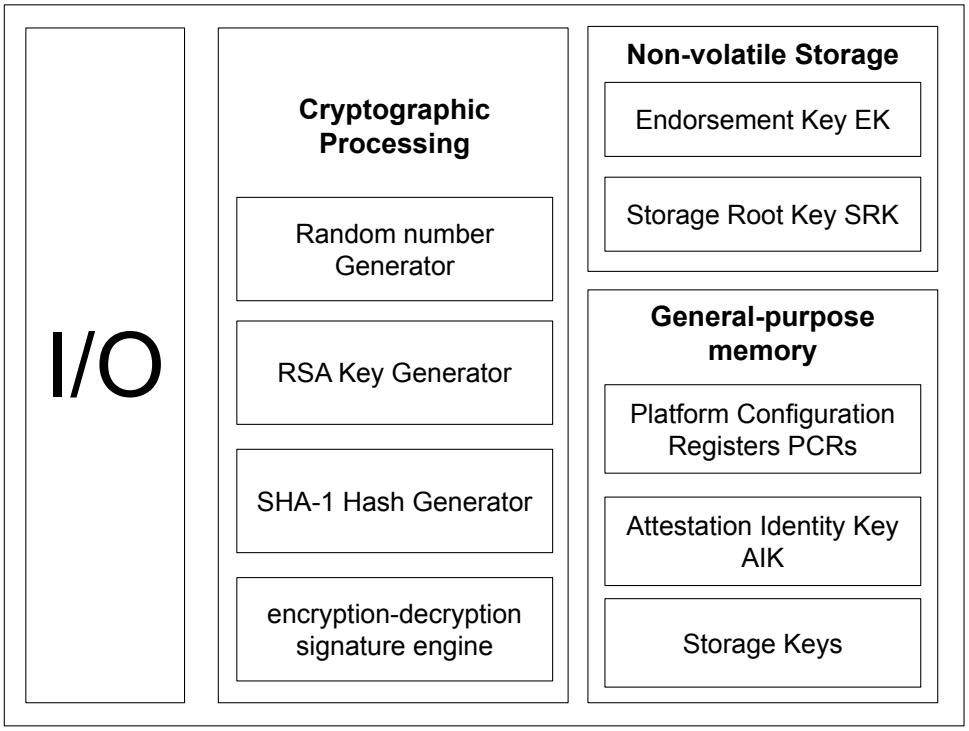

Figure 2.2. TPM summary schema

2.4.2 Implementations

- **Discrete TPM**: a dedicated chip that implements TPM functionality in its own tamper-resistant semiconductor package.
- **Integrated TPM**: hardware of the TPM embedded in another chip, in this case, is not required that the TPM implements tamper resistance
- **Firmware TPM**: run the firmware corresponding to the TPM in the CPU's TEE (Trusted Execution Environnement).
- **Hypervisor TPM**: virtual TPM provided by a hypervisor, has a security comparable with the firmware TPM.
- **Software TPM**: an emulator of a TPM, runs in userspace, used only in the development process.

2.4.3 key hierarchies

- **Platform Hierarchy**: the platform is the board or system that hosts the TPM, and manages key associated with platform-specific operations.

- **Endorsement Hierarchy**: this hierarchy includes the Endorsement Key (EK), which is unique to each TPM and used for identity and attestation purposes.
- **Storage Hierarchy**: manages keys used for data storage and retrieval within the TPM.

2.4.4 Securely storing data

The TPM can be utilized to securely store data, providing two types of isolation: physical and cryptographic. Physical isolation involves storing data in the non-volatile RAM of the TPM, which has limited space and incorporates mandatory access control. This makes it difficult to easily access or remove the secure data.

In cryptographic isolation, the key is stored outside the TPM but encrypted with a key stored in the non-volatile RAM. This approach allows for unlimited storage capacity from the TPM's perspective. The encrypted data can only be decrypted by the specific TPM that holds the corresponding key, preventing any data migrations. Mandatory access control is still enforced in this scenario as well.

2.4.5 TPM Platform Configuration Register PCR

The TPM can report the present system state by utilizing a special set of registers known as PCR. These registers store and maintain the history of the platform configuration. The set of PCRs serves as the fundamental mechanism for recording and verifying the integrity of the platform. These registers are exclusively reset during a platform reset, which occurs after a reboot or hardware signal. Resetting the registers entails restoring them to a value of 0. This is crucial because even if malicious code manages to infect the system, it cannot manipulate the registers to revert to a specific value.

The Platform Configuration Register [9] serves as an internal register within the TPM and holds a 20-byte digest, representing the software and hardware configuration of a host system. This digest is established before system deployment and continuously monitored throughout the system's lifespan to track any alterations in its hardware and software setup. In case the system detects any modifications to its configuration or the corresponding PCR, it can take appropriate actions. Consequently, the system possesses the capability to safeguard itself from potential malicious attacks.

These registers have only two operations: one is "reset" and the second one is "extend" 2.1. The PCR operates by hashing the previous value stored in it, combined with the digest of new data. The resulting hash becomes the updated value of the PCR. This operation is limited to hashing and updating the PCR value.

$$PRC_{new} = hash(PCR_{old}||digest\_of\_new\_data) \tag{2.1}$$

Each PCR has a different scope, the first ones are used for the services launched at boot. Usually are divided in this way:

2.4.6 TPM versions

The TPM specifications have evolved over time, resulting in different TPM versions. The major TPM versions are:

- **TPM 1.2**: was the initial version of the TPM specification released by the Trusted Computing Group (TCG). It introduced basic security features and capabilities, including key generation, storage, and cryptographic operations. TPM 1.2 chips were commonly used in older systems but are gradually being replaced by newer versions.

- **TPM 2.0**: is the current and most widely adopted version of the TPM specification. It offers significant improvements over TPM 1.2, including enhanced cryptographic algorithms, command sets, and security features. TPM 2.0 provides more flexibility, better remote attestation capabilities, and improved support for modern cryptographic standards. TPM 2.0 chips are commonly found in modern computing devices.

- **TPM 2.0 with Firmware TPM (fTPM)**: TPM 2.0 introduced Firmware TPM (fTPM) as an alternative to dedicated hardware TPM chips. fTPM utilizes a software implementation that operates within the system's firmware, making use of the existing hardware components instead of requiring a separate TPM chip.

TPM 2.0 maintains backward compatibility with TPM 1.2, enabling systems with TPM 2.0 to function with software intended for TPM 1.2. Nonetheless, it's crucial to acknowledge that there can be variances in features and capabilities between the two versions.

PCR	PCR Usage
0	SRTM, BIOS, Host Platform Extensions, Embedded Option, ROMs and PI Drivers
1	Host Platform Configuration
2	UEFI driver and application Code
3	UEFI driver and application Configuration and Data
4	UEFI Boot Manager Code and Boot Attempts
5	Boot Manager Configuration and Data and GPT/Partition Table
6	Host Platform Manufacturer Specific
7	Secure Boot Policy
8-15	Defined by the static OS
16	Debug
23	Application Support

Table 2.1. Standard PCRs usage in a system. The SRTM indicates the Static Root of Trust for Measurements

2.4.7 Check TPM status

Firstly to check the activation of the TPM access the BIOS menu and navigate to the security section. On some UEFI bios, the TPM options are under the Advanced section. To check the status depends on the Operative system on the machine:

- **Windows**: under the Device Security option looking in the `Security Processor details` we can see the information about the TPM.

- **Linux**: the activity of a TPM can be checked using:

  ```
  $ sudo dmesg | grep -i tpm | grep -i
  ```

  Reserving the output will look like:

  ```
  [0.007734] ACPI: Reserving TPM2 table memory [mem 0x41ac6000-0x41ac604b]
  ```

  In this example the TPM is active and the system reserves a memory range `(0x41ac6000-0x41ac604b)` for the TPM2 table during the ACPI (Advanced Configuration and Power Interface) initialization process. TPM devices are installed in the path `/dev/tpmX`.

2.4.8 Widespread adoption of TPM

The global adoption of TPM [10] encountered various challenges, including the initial absence of standardized implementations and compatibility issues. However, significant progress has been made in addressing these concerns. Incorporating TPM into systems may necessitate hardware upgrades or investments in new devices that support TPM capabilities. The cost factor can pose a significant obstacle for organizations with limited budgets or resource constraints. To overcome this challenge, exploring cost-effective solutions and considering retrofitting existing systems to incorporate TPM functionalities is advisable.

Another solved problem regards the `off by default` status of the TPM 1.2 in the user's computer. the user is required to activate it via firmware menus manually. Because most users never touch their firmware settings, most TPMs have never been turned on. TPM 2.0 fixes these problems by ensuring that the TPM is on by default and by creating a platform entity, ensuring that platform firmware has full access to TPM resources.

Chapter 3

IMA Integrity Measurement Architecture

Ensuring robust security systems often requires more than event logs. Linux systems offer a range of open-source tools that effectively enhance file integrity, one of them is IMA [11]. The Integrity Measurement Architecture (IMA), a component of the Linux kernel's integrity subsystem, plays a vital role in this regard. By calculating hashes of files and providing reporting capabilities, IMA strengthens the detection of anomalies and supports the hunt for intruders. The integrity subsystem, supplemented by the Extended Verification Module (EVM), detects tampering with offline security attribute extensions, such as "SELinux", which form the basis for clearance decisions in the Linux Security Modules (LSM) framework. By means of `IMA hooks`, IMA is able to perform actions before an object is used. In this way is possible to measure a file before its inode is accessed.

IMA extends the concept of Secure Boot to the entire Linux OS. It introduces kernel-level hooks to collect file hashes before execution, modification, or reading. Collected hashes can be appraised locally or remotely via remote attestation. If a TPM Chip is present, IMA can directly extend the collected measure on the TPM, typically on PCR10, creating a Root of Trust and enabling integrity verification of the Measurements List.

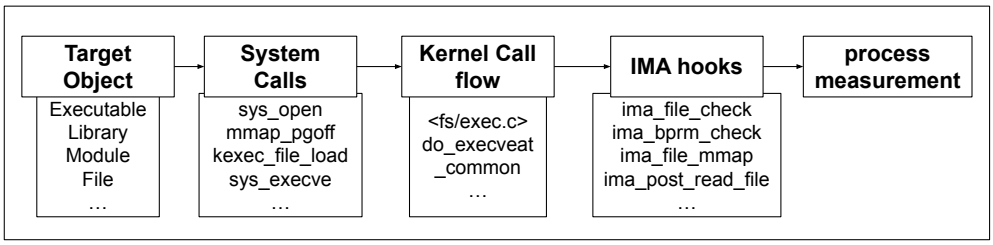

Figure 3.1. IMA hooks schema

3.1 Enable Linux IMA

In order to activate Linux IMA the following `.config` options have to be enabled:

```
CONFIG_INTEGRITY=y
CONFIG_IMA=y
CONFIG_IMA_MEASURE_PCR_IDX=10
CONFIG_IMA_LSM_RULES=y
CONFIG_INTEGRITY_SIGNATURE=y
```

```
CONFIG_IMA_APPRAISE=y
```

Listing 3.1. Kernel modules necessary to enable IMA

A value can be enabled:

- **at runtime**: using the command `sysctl` is possible to define new values for the parameters.
- **compiling the kernel**:
  1. `make menuconfig` where is possible to enable kernel modules;
  2. `make` to compile the kernel;
  3. `make modules` to compile the kernel modules;
  4. `sudo make modules_install` to install the kernel modules;
  5. `sudo make install` to install the compiled kernel and add it to the boot menu.

3.2 Components

- **IMA-Measurement**: IMA measurement expands upon the concept of Measured Boot and incorporates it into the operating system. This extension facilitates Remote Attestation, where a third party challenges the system to validate its integrity. To achieve this, IMA maintains a sequential list of hash values within the kernel. This list encompasses all files that have been measured since the OS assumed control. While IMA can function independently of a Trusted Platform Module, it is commonly utilized alongside a TPM to establish a more robust and hardware-rooted chain of trust. Subsequently, the measurement list becomes a valuable asset in the process of remote attestation. This process involves presenting a comprehensive record of all software executions to an external entity conducting an investigation. The remote party can then cross-reference this list with a database containing established, valid reference measurements. Any disparities from the intended state of integrity can be identified through this comparative analysis.
- **IMA-Appraisal**: the IMA Appraisal system expands the Secure Boot principle to encompass both the operating system and its applications. It accomplishes this by incorporating known valid hash values for each file into the extended attributes of those files. These hash values serve as reference points for the Integrity Measurement Architecture, which compares them to the actual measurements of each file. Should a disparity arise between the measured and expected hash values, IMA prevents access to the file. In addition to merely storing hash values within the extended file attributes, the option exists to utilize digital signatures. The IMA appraisal process offers four distinct operational modes:
  - `Fix mode`: this mode gathers and appends the measurements as valid hash values to the extended attributes of the file.
  - `Log mode`: mismatches between the measured hash and the hash in the file's extended attributes are logged.
  - `Enforce mode`: access is denied if a mismatch between measured and expected hash values is detected.
  - `Off mode`: this mode disengages the IMA appraisal mechanism.
- **IMA-Audit**: insert the hash calculated of a file in the system logs, which may be useful for forensic analysis for example.

3.3 Policies

IMA Policies establish the criteria for selecting files to undergo measurement. IMA incorporates three predefined policies that are conveyed to the kernel:

- **Tcb Policy**: this policy facilitates the measurement of various components, including kernel modules, executed software, files loaded into memory for execution via `mmap`, and files opened for reading by the root user.
- **Appraise_tcb Policy**: in addition to measuring, the `appraise_tcb` policy also conducts in-place appraisals of the same components covered by the tcb policy. Access is denied if files fail to align with their verified hash values.
- **Secure_boot Policy**: The `secure_boot` policy exclusively focuses on the appraisal of the kernel, its associated modules, and the IMA policies themselves.

The policy can primarily be defined in two distinct manners:

1. Specify a standard policy from the kernel command (adding it to the grub command line) example of inserting policy from the command line:
   - `$ vim /etc/default/grub` to modify the grub configuration file;
   - change the value of `GRUB_CMDLINE_LINUX` inserting the value `ima_policy=tcb` for example;
   - to update the grub configuration `$ sudo update-grub`
   - reboot and check the value of the kernel command line using: `$ cat /proc/cmdline`.

   Other examples of parameters that could be inserted are:

   `lsm=integrity, ima_appraise=enforce, ima_policy=tcb`
2. Append a valid policy to the file `policy` under the ima folder (note that the insertion in the file policy has to be enabled in the kernel configuration:

   `$ echo "measure func=MMAP_CHECK mask=MAY_EXEC" > \`

   `/sys/kernel/security/integrity/ima/policy`

   A valid policy is composed by:

```
action: measure | dont_measure | appraise | audit |
    dont_appraise | hash | dont_hash

condition:= base | lsm [option]
    base: [[func=] [mask=] [fsmagic=] [gid=] [egid=]
        [fsuuid=] [fsname=] [uid=] [euid=]
        [fowner=] [fgroup=]]

    lsm: [[subj_user=] [subj_role=] [subj_type=]
        [obj_user=] [obj_role=] [obj_type=]]

    option: [digest_type=] [template=] [permit_directio]
        [appraise_type=] [appraise_flag=]
        [appraise_algos=] [keyrings=]
```

Listing 3.2. Structure of a IMA policy

3.4 Templates

The initial IMA template [12] has a predetermined size, encompassing both the hash of the file data and its corresponding pathname. The file data hash is confined to a maximum of 20 bytes. Meanwhile, the pathname is represented as a null-terminated string, with a restriction of 255 characters in length. To overcome these constraints and introduce supplementary file metadata, an expansion of the current IMA version is imperative, involving the definition of additional templates.

Nonetheless, the primary challenge in implementing this capability is that every time a new template is established, the functions responsible for generating and presenting the list of measurements would need to incorporate code to manage the new format. Consequently, this would lead to substantial growth in these functions over time. At the heart of this solution lies the creation of two novel data structures: a template descriptor, which defines the elements to be included in the measurement list, and a template field, which facilitates the generation and display of specific data types.

To accommodate a new data type, the process involves generating and presenting measurement entries. Introducing a fresh template descriptor mandates outlining the template's format through the `ima_template_fmt` kernel command line parameter. During system boot-up, IMA initializes the designated template descriptor by converting the format into an array of template field structures selected from the pool of supported options.

Following the initialization phase, IMA engages `ima_alloc_init_template()` to forge a novel measurement entry using the chosen template descriptor. This stage particularly underscores the advantages of the new architecture: the said function is devoid of specific code tailored to individual templates. Instead, it directly invokes the `init()` method associated with the template fields linked to the selected template descriptor, subsequently saving the outcome (inclusive of the pointer to allocated data and data length) within the measurement entry structure.

3.5 Measure Option

It allows the system to measure and verify the integrity of files and processes at various points in their lifecycle. The `measure` option in the IMA defines the rules for measuring files or processes. When a file or process is accessed or executed, the IMA subsystem can generate a cryptographic hash (usually `SHA1` or `SHA256` hash) of the file or process, which is then stored in the IMA measurement list.

The measure option provides flexibility in determining which files or processes should be measured and how the measurement is performed. It allows you to define specific criteria, such as the path of the file or the characteristics of the process, to decide whether to measure or skip measuring certain entities. By utilizing the measure option effectively, you can enforce integrity measurement policies and ensure that critical files and processes on your Linux system remain unchanged and uncorrupted. This helps detect and prevent tampering or unauthorized modifications, providing an additional layer of security for your system.

Examples of measure and don't measure policies:

```
dont_measure fsmagic=SELINUX_MAGIC
measure func=BPRM_CHECK
```

3.6 Measured Boot

The first entry in the measurement list is the Boot aggregate, the hash derived from the measured boot. Measured Boot is a security feature designed to ensure the integrity of the boot process in a computer system. It verifies boot components, like firmware, bootloader, and operating system, to detect tampering or unauthorized modifications. Measured Boot is an additional security protocol that entails recording measurements in TPM's PCRs during the boot process, while still permitting the completion of the boot sequence. Subsequently, these measurements can be retrieved from the TPM, and an Audit log containing all the measurements is also accessible. Consequently, Measured Boot abstains from rendering an assessment of the boot stages' integrity. Rather, it presents an avenue for an external verifier to meticulously examine the TPM Audit Log and PCR values. This scrutiny enables the verifier to discern the components that underwent measurement, ascertain their specific measurements, and subsequently arrive at an informed decision. The collection of measures is done at different levels:

- initial stage of the boot loader, which is considered trustworthy, forms the core root for measurement;
- computes the hash of the second stage boot loader UEFI/BIOS;
- measures the operating system and then loads it for execution.

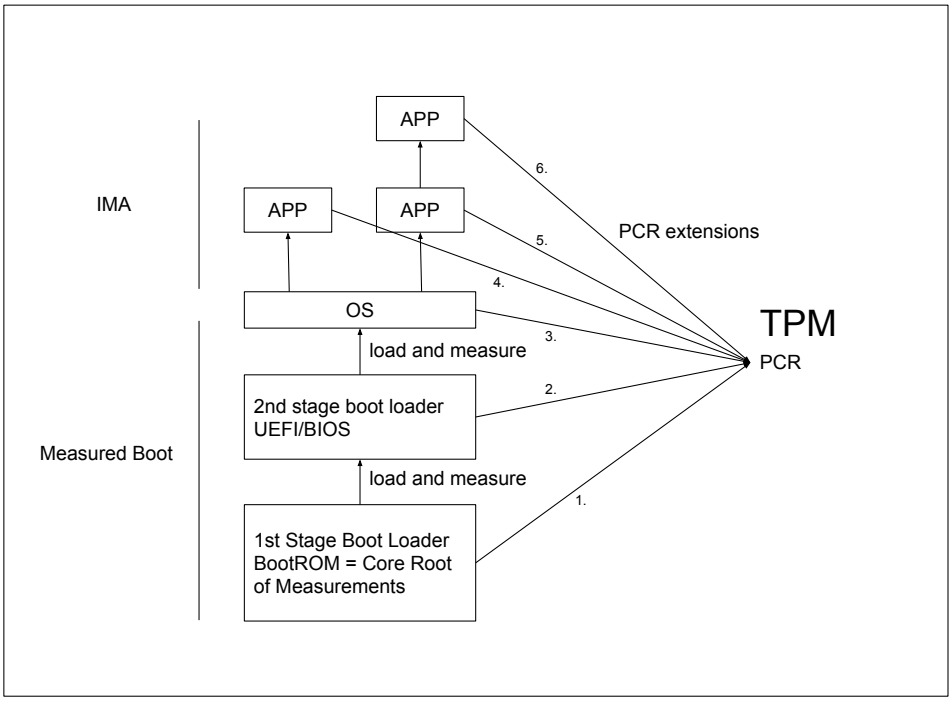

Figure 3.2. Boot Aggregate and IMA schema

3.7 Remote Attestation

Remote attestation involves a systematic procedure where a designated system, known as the attestor, furnishes cryptographic evidence, in the form of signed data, regarding all executed software and accessed files. This evidence is then transmitted to an examining system, referred to as the verifier, which undertakes the task of authenticating the provided evidence. The outcome of this authentication culminates in a cryptographic endorsement certifying the integrity of the attestor system.

For remote attestation to be feasible, the attestor system must possess a reliable Measured Boot mechanism. This is commonly achieved by implementing a hardware configuration that incorporates Real-Time Monitoring and a Trusted Platform Module, coupled with boot components that are TPM-aware.

IMA supports the remote attestation, since is a long operation a nonce in the request is used to avoid replay attacks. The use of a TPM in remote attestation ensures the integrity and trustworthiness of the target device's platform, enabling secure communication and establishing a foundation for establishing trust between remote devices. the attestation follows these steps:

1. **attestation Request**: the remote device initiates the attestation process by sending a request to the target device that contains the TPM;

2. **quote Generation**: the target device's TPM creates a quote, which is a digitally signed measurement of the system's current state;
3. **attestation Response**: the target device sends the generated quote back to the remote device;
4. **verification**: the remote device validates the received quote by verifying its integrity and authenticity. This involves checking the digital signature using the TPM's public key;
5. **trust Assessment**: the remote device compares the verified quote against a known trusted baseline or policy. If the quote matches the expected values, the target device is deemed trustworthy;
6. **attestation Result**: the remote device generates an attestation result based on the trust assessment, indicating whether the target device is trusted or not.

3.8 Securityfs and IMA files

The `securityfs` is meant to be used by security modules and is mounted on `/sys/kernel/security`, it has three main API functions, one for creating a file, one for creating a directory, and a generic function for remove a file or a directory. When creating a file a struct `file_operations` is passed as a parameter, it contains the functions pointer for interacting with the file.

In the securityfs under the path `/sys/kerel/security/integrity/ima` there are the IMA files:

- `policy`: policies active at the moment.
- `ascii_runtime_measurements`: list of measures includes PCR extended, template hash, name of the template, and the template fields, in this case, the measure hash and the path of the file measured.

  `10 fd..ac ima-ng sha1:8d..30 /bin/ls`
- `binary_runtime_measurements`: same as `ascii_runtime_measurements` but in binary format.
- `runtime_measurements_count`: number of measurements collected.
- `violations`: number of violations, a violation occurs when the integrity of a file or process is compromised, indicating that the file or process has been modified or tampered.

3.9 Code Structure

3.9.1 Store the measurements

The function `ima_store_measurement` is responsible for storing the measurement of a file. It creates an IMA template and then calls `ima_store_template` to store the template. This function is only reached if the inode has not been measured before, the inode-associated integrity data are checked, but there are scenarios where the measurement could already exist, such as multiple copies of the same file on different filesystems or if the inode was previously flushed along with the hashing information stored in the `iint` structure. It is important to note that this function must be called with the `iint-mutex` held.

```
void ima_store_measurement(struct integrity_iint_cache *iint, struct file
    *file, ...) {
    if (iint->measured_pcrs & (0x1 << pcr) && !modsig)
        return;
    result = ima_alloc_init_template(&event_data, &entry, template_desc);
```

```
    result = ima_store_template(entry, violation, inode, filename, pcr);

    if ((!result || result == -EEXIST) && !(file->f_flags & O_DIRECT)) {
        iint->flags |= IMA_MEASURED;
        iint->measured_pcrs |= (0x1 << pcr);
    }
}
```

Listing 3.3. IMA function that handles the store of a measurement

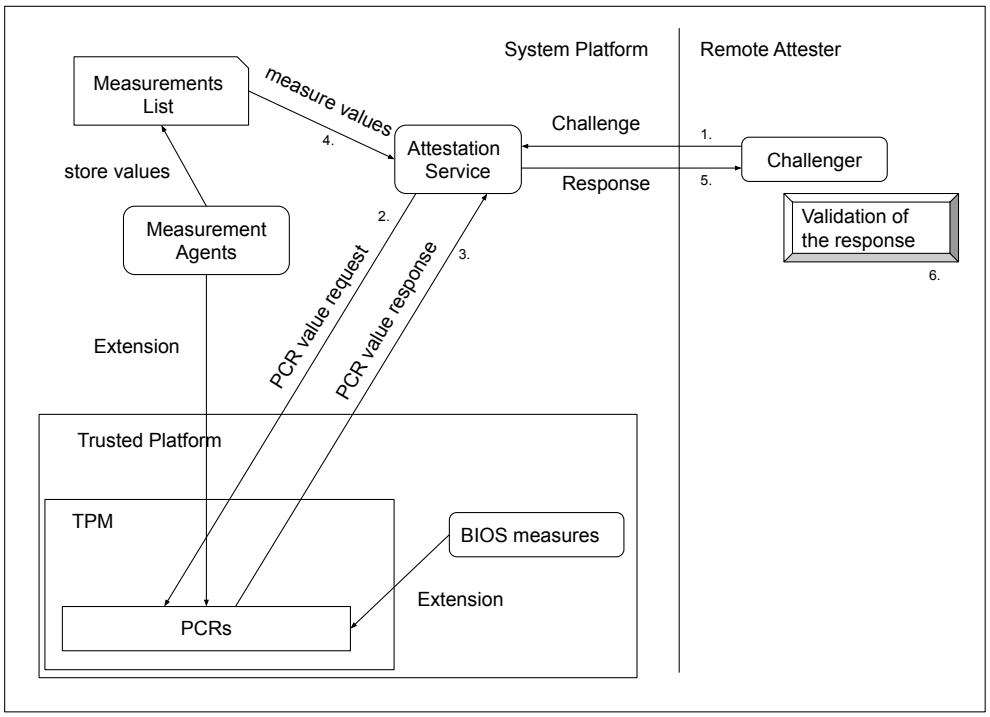

Figure 3.3. IMA summary schema

3.9.2 Policy matching

This function controls the list of policies looking for a policy that matches the hook identifier `func`. The decision regarding measurement is determined by the conditions of `func/mask/fsmagic` and `LSM (subj/obj/type)`. In case matches a policy saves the PCR that will be used and the template descriptor.

To accommodate multiple updates to the IMA policy, it is necessary to lock the list when traversing it. Given that reads significantly outnumber writes, the `ima_match_policy()` function is classical RCU (Read-Copy-Update). RCU is a synchronization mechanism used in concurrent programming to allow efficient and safe access to shared data in situations where the majority of operations are read, and updates are infrequent. The basic idea behind RCU is to allow multiple threads to concurrently read shared data without any locks or blocking, while still providing a consistent and up-to-date view of the shared data. RCU achieves this by delaying the actual update of the shared data until all the active readers have completed their access.

```
int ima_match_policy(struct mnt_idmap *idmap, struct inode *inode, ...) {
    rcu_read_lock();
    ima_rules_tmp = rcu_dereference(ima_rules);
    list_for_each_entry_rcu(entry, ima_rules_tmp, list) {
```

```
        if (!ima_match_rules(entry, idmap, inode, cred, secid, func, mask,
            func_data))
            continue;
        action |= entry->flags & IMA_NONACTION_FLAGS;
        action |= entry->action & IMA_DO_MASK;
        if ((pcr) && (entry->flags & IMA_PCR))
              *pcr = entry->pcr;
        if (template_desc && entry->template)
              *template_desc = entry->template;
        ...
    }
    rcu_read_unlock();
        ...
    return action;
}
```

Listing 3.4. IMA function that controls the policy regarding the hook

3.9.3 Hash Table

A hash table is a popular data structure employed in computer science to achieve efficient storage and retrieval of data. By utilizing a method called hashing, it enables rapid insertion, deletion, and lookup operations. This grants the advantage of swift data manipulation and retrieval. It is implemented as part of the kernel's infrastructure and is used for various purposes.

Is implemented as a fixed-size array, and the collisions are handled using separate chaining, that stores multiple entries with the same hash value in an array. To retrieve or store data within a hash table, a key is supplied, which is hashed to determine the slot where the corresponding value may be stored. The key for the IMA hash table is the Template entries hash.

The struct of the IMA hash table is implemented using the standard implementations for linked lists as follows:

```
struct ima_h_table {
    atomic_long_t len; /* number of stored measurements in the list */
    atomic_long_t violations;
    struct hlist_head queue[IMA_MEASURE_HTABLE_SIZE];
};
```

Listing 3.5. IMA hash table struct

When a new hash is calculated is checked if present in the hash table. If not inserted otherwise returns the error `-EEXIST`. The hash table can be disabled at the kernel level, This can be done by enabling the option `CONFIG_IMA_DISABLE_HTABLE`.

3.9.4 Integrity data associated with an inode

The integrity cache, an integral component of the integrity filesystem, serves the purpose of tracking whether a file has been measured and the corresponding PCR to which it has been extended. When a file is measured, its measurement is stored in the `ima_hash`. Subsequently, when there is a need to store the measurement, it is retrieved.

```
struct integrity_iint_cache {
    struct rb_node rb_node; /* rooted in integrity_iint_tree */
    struct mutex mutex; /* protects: version, flags, digest */
    struct inode *inode; /* back pointer to inode in question */
    u64 version;  /* track inode changes */
    ...
    enum integrity_status evm_status:4;
```

```
    struct ima_digest_data *ima_hash;
};
```

Listing 3.6. Integrity data associated to a inode struct

3.9.5 Process Measurement

When an event triggers the IMA hooks, the first step is to extract the action to be executed from the policy and file information. Depending on the obtained action, different functions are called. Before performing a measurement, the integrity-related data is checked to determine if the file has already been measured in the same PCR. If the file is already measured, the measurement process is skipped. The `collect measurement` function calculates the hash and stores it in the associated integrity data.

```
static int process_measurement(struct file *file, char *buf, ...) {
    action = ima_get_action(file_mnt_idmap(file), inode, ...);
    ...
    iint->flags |= action; action &= IMA_DO_MASK;
    action &= ~((iint->flags & (IMA_DONE_MASK ^ IMA_MEASURED)) >> 1);

    if ((action & IMA_MEASURE) && (iint->measured_pcrs & (0x1 << pcr)))
        action ^= IMA_MEASURE;
    ...
    hash_algo = ima_get_hash_algo(xattr_value, xattr_len);
    rc = ima_collect_measurement(iint, file, buf, size, hash_algo, modsig);
    if (action & IMA_MEASURE)
        ima_store_measurement(iint, file, pathname, xattr_value, xattr_len,
            modsig, pcr, template_desc);

    if (action & IMA_AUDIT)
        ima_audit_measurement(iint, pathname);
    ...
    return 0;
}
```

Listing 3.7. IMA function that initiates the measure process

3.9.6 Templates

For optimal adaptability, every entry within the measurement list is subdivided into distinct fields. Each of these fields is equipped with its own unique set of functions to manage operations effectively. A template is then meticulously assembled from a carefully chosen assortment of fields. This approach not only provides a high degree of flexibility but also opens avenues for the creation of fresh templates. Importantly, each template can incorporate a variable count of fields, and the nature of each field can vary independently.

```
static struct ima_template_desc builtin_templates[] = {
    {.name = IMA_TEMPLATE_IMA_NAME, .fmt = IMA_TEMPLATE_IMA_FMT},
    {.name = "ima-ng", .fmt = "d-ng|n-ng"},
    ...
    {.name = "", .fmt = ""}, /* placeholder for a custom format */
    };

    static const struct ima_template_field supported_fields[] = {
    {.field_id = "d", .field_init = ima_eventdigest_init,
     .field_show = ima_show_template_digest},
    {.field_id = "n", .field_init = ima_eventname_init,
```

```
        .field_show = ima_show_template_string},
    ...
};
```

Listing 3.8. IMA templates and fields structs

3.10 Threats

Assuming an attacker possesses the capability to either introduce a malicious block device into an IMA-protected system or modify the firmware of an existing block device. A significant vulnerability of IMA arises from its susceptibility to a "TOCTOU" (Time-of-Check to Time-of-Use) attack [11]. When Linux executes a file, it must calculate its hash beforehand. However, if the file is too large to fit into the operating system's buffer cache, exceeding the machine's available RAM, it will be read twice. The first read is performed by IMA to compute the hash value, and the second read is carried out for execution. During this second read, IMA does not re-measure the file. Consequently, IMA remains unaware if the block device supplies a modified version of the file during the second read.

The current implementation of IMA in the Linux kernel falls short in terms of effectiveness. Simply measuring a file once is not reliable when considering attacks on block device firmware, especially since a block device can unilaterally modify a file. Therefore, it is crucial to assess mechanisms like `dm-verity` and `fs-verity` to determine if they can address this issue. In fact, the community has been discussing the possibility of integrating IMA with `fs-verity` mechanisms, enabling the verification of pages as they are accessed.

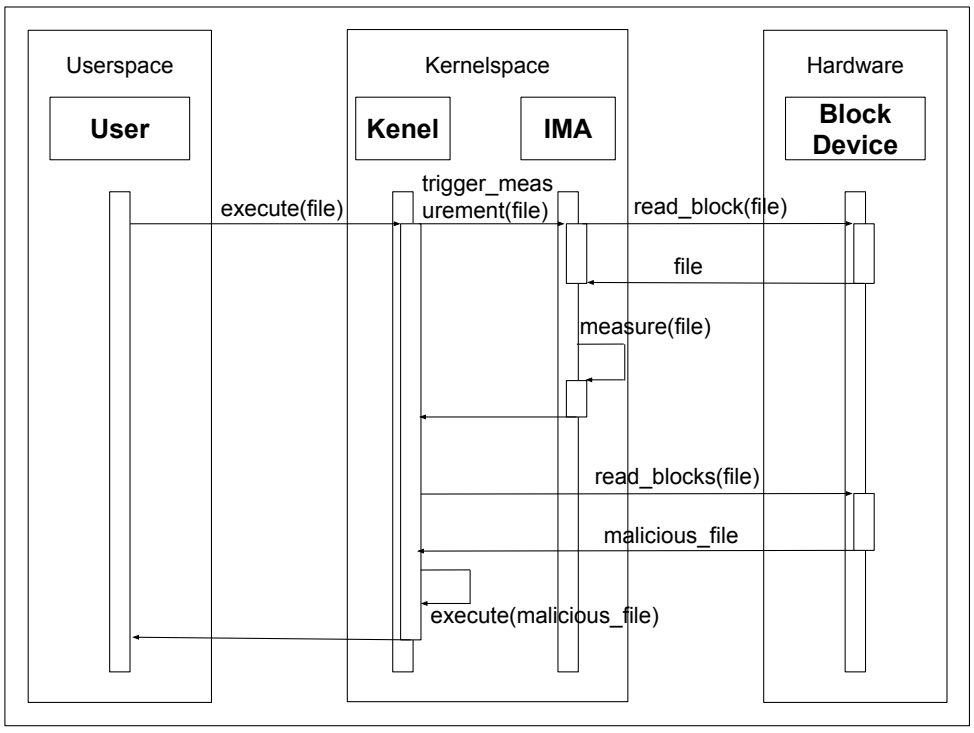

Figure 3.4. IMA attack schema

3.11 Scenarios of using Linux IMA

These case studies illustrate the versatility and benefits of implementing IMA in different contexts, emphasizing its role in enhancing the security, integrity, and trustworthiness of computing systems across diverse industries and use cases:

- **Cloud Security**: IMA has been employed in cloud computing environments to enhance the security and integrity of virtual machine (VM) instances. By enabling IMA and configuring appropriate policies, cloud providers can ensure that the VM images and application binaries running within the instances remain unaltered and trustworthy. IMA measurements can be used for attestation, allowing cloud users to verify the integrity of their VMs and ensure they are running in a trusted environment.
- **Embedded Systems and Internet of Things (IoT)**: IMA is increasingly utilized in embedded systems and IoT devices to protect against firmware or software tampering. By measuring and appraising critical components such as bootloaders, kernel images, and device firmware, IMA ensures that these components remain intact and unmodified. This helps mitigate the risk of malicious actors tampering with the embedded system's software stack and improves the overall security of IoT deployments.
- **Digital Forensics and Incident Response**: IMA has proven valuable in digital forensics and incident response investigations. By leveraging the measurements recorded by IMA, forensic analysts can establish a baseline of trusted files and detect any unauthorized changes or malicious activities. This enables investigators to identify potential security breaches, analyze the impact of the incident, and track the progression of an attack through the compromised system.
- **Software Supply Chain Security**: IMA plays a crucial role in ensuring the integrity of software packages throughout the supply chain. By signing software packages with a trusted key and measuring the binaries, IMA provides a mechanism to verify the integrity of the packages at each stage of the supply chain. This helps prevent tampering or unauthorized modifications, thereby enhancing the security and trustworthiness of software distributions.
- **Compliance and Regulatory Requirements**: IMA can assist organizations in meeting compliance and regulatory requirements in various industries. For instance, in sectors such as healthcare or finance, where data integrity and security are critical, IMA can help demonstrate that systems and software remain unaltered, providing evidence of compliance with regulatory standards. IMA's ability to attest to the integrity of critical components is valuable for audits and compliance assessments.

Chapter 4

Virtualization of PCRs for Containers

The central concept emphasized in this approach [13] prioritizes safeguarding privacy. A crucial aspect of this initiative involves preventing the disclosure of information regarding the underlying host and other users' containers to a remote verifier. This project addresses this concern by partitioning the measurement log (ML), a step that not only ensures privacy but also reduces attestation latency. To provide robust protection, each ML is secured using a hardware-based root of trust, known as a cPCR (Container-based Platform Configuration Register). For this project is important to ensure the privacy of the underlying host and other user's containers. Due to IMA's practice of measuring all components into a single PCR and recording them in a unified Measurement Log, every verifier must gather the complete ML when assessing the integrity of containers. This setup poses a risk as each verifier gains access to all the prover's information. As a result, adversaries can exploit this situation to identify vulnerabilities in the prover and even extract information from other users' containers. Key contributions are:

- attestation of a specific container without revealing any information about other containers or unnecessary details of the underlying host to the remote verifier. This is particularly important in scenarios where a malicious verifier might be aware of the components present in the underlying host and other containers. The goal is to maintain strict privacy and only provide essential information related to the designated container during the attestation process;
- gather integrity evidence encompassing all components and dependencies of the designated container. To achieve this, the use of a container-based PCR (cPCR) is proposed to guarantee the preservation of privacy and integrity evidence through a hardware-based Root of Trust (RoT). This ensures that any tampering or manipulation of the evidence will be detectable by the remote verifier;
- decreasing the latency of container attestation.

4.1 Architecture Overview

IMA has established Remote Attestation as a means for a verifier to verify the integrity of a prover. Upon receiving an attestation request, the Attestation Agent within the prover gathers integrity evidence, which includes PCR values, the signature signed by the TPM, and the Measurement Log. Transmitting the complete ML to the verifier raises privacy concerns in a container environment. In the case where the verifier is the owner of a specific container, they should not have access to unnecessary information about other containers or the underlying host. Therefore, ensuring privacy during Remote Attestation becomes crucial to address these issues. The Linux namespace mechanism is used to create isolated environments for containers. This mechanism partitions

system resources into distinct instances across various aspects, such as mount, hostname, IPC, PID, and network. As a result, each container typically operates within its unique namespace, ensuring effective isolation from other containers.

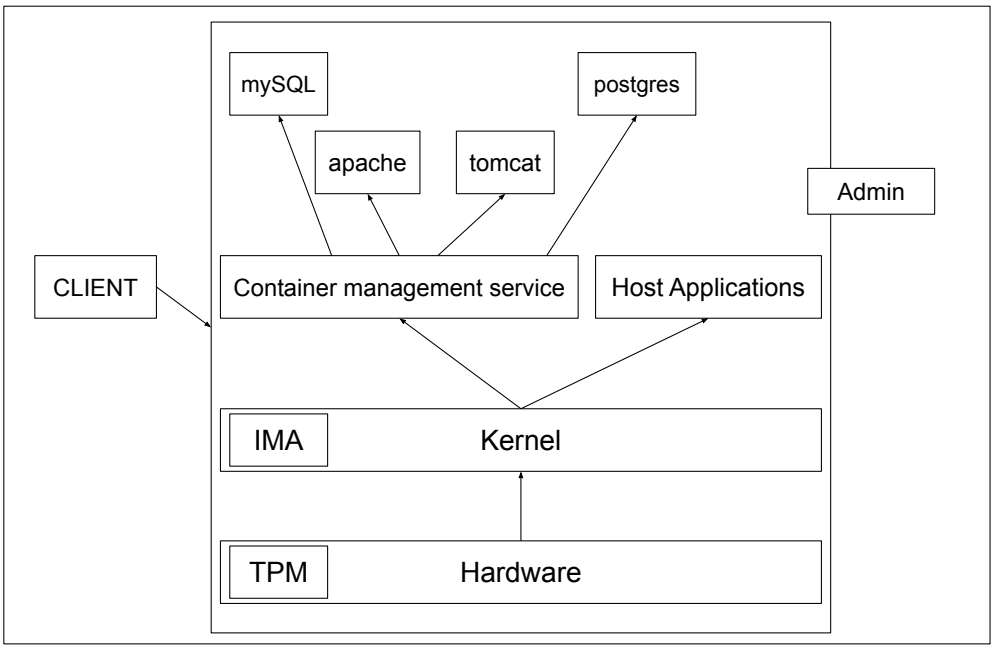

Figure 4.1. Architecture schema

In the traditional approach, trusted boot [14] allows for the measurement of components during a platform's boot process. This is achieved through a measure-before-loading mechanism, which constructs a Chain of Trust and transfers trust from the Root of Trust to the OS kernel. The measurement mechanisms change once trust reaches the application layer. At this stage, all software components within the application layer are measured by the Integrity Measurement Architecture and recorded in the Measurement Log based on their loading time. Consequently, the sequence of loaded software components is integrated into the Chain of Trust within the application layer. The chain of trust can be decomposed into the following partitions:

- **Integrity of prover's Boot Time**: starts by turning on the computer, followed by the initialization of the BIOS. Afterward, the GRUB bootloader takes over, and finally, the OS kernel is loaded, ending with the successful booting of the operating system.
- **Integrity of Containers' dependencies**: refers to the services responsible for container management and the corresponding files or libraries they rely on.
- **Integrity of a Container's Boot Time**: refers to the images and boot configurations utilized by container management services when they launch a container.
- **Integrity of a Container's Applications**: begins with the successful launch of a container by container management services and concludes with its subsequent shutdown. This process encompasses all the components operating within the container.
- **Integrity of Host Applications**: starts with the successful launch of the OS kernel and concludes with the subsequent shutdown of the prover. Please note that this partition excludes any mention of container management services and the containers themselves.

Referring to Figure 4.1, from a container's viewpoint, its direct dependency lies with the container management service. Other host applications remain isolated from containers through namespaces. The privacy requirement addressed in this solution is as follows: The integrity

evidence sent back from the prover to the verifier must not disclose any information about the applications running on the host or in other containers. When a verifier requests to attest a container, the prover can combine these partitions to demonstrate its trustworthiness. However, components that do not belong to the Chain of Trust of a designated container should remain undisclosed to the verifier. A verifier cannot distinguish whether his container is the only container running on the prover.

4.2 New components

Here is the architecture schema Figure 4.2 encompassing three additional components: the Split Hook, the namespace parser, and the container PCR. Container-based virtualization relies on the namespace mechanism to isolate system resources from one another, necessitating that each container has its distinct namespace.

- **Split Hook**: examines the system call to generate a new namespace and then informs the kernel about the event to split ML.
- **Namespace Parser**: retrieves the namespace number of the current process to determine which partition the current ME belongs to
- **container-PCR**: responsible for safeguarding each ML partition with TPM involves the maintenance of cPCRs. cPCRs are data structures within the kernel that simulate PCR.

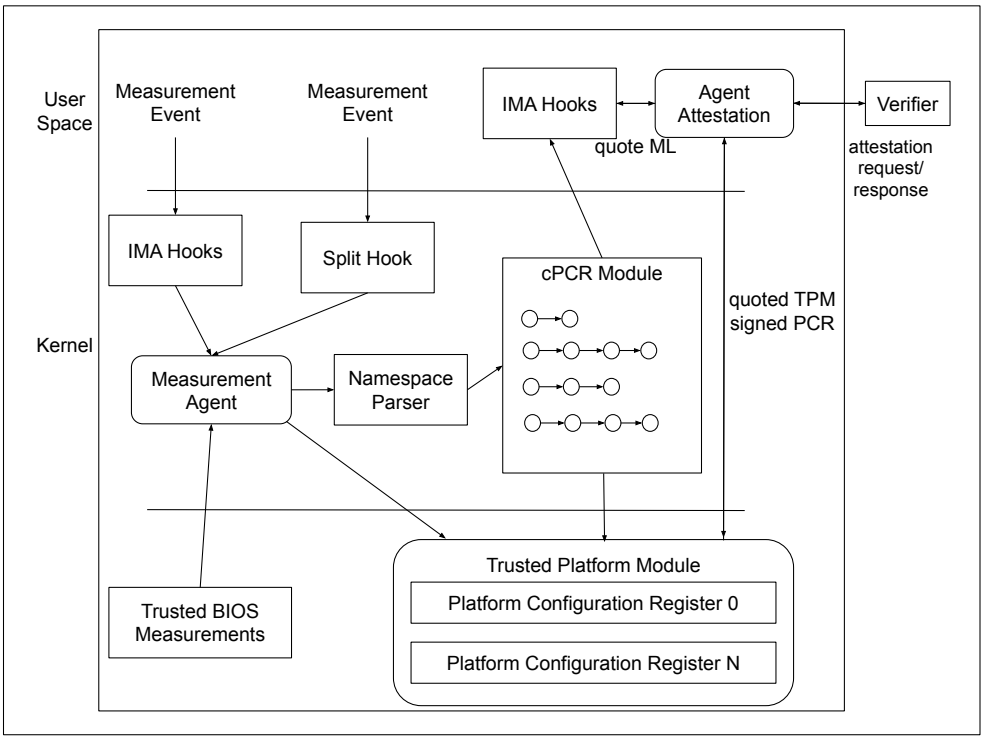

Figure 4.2. Complete schema [13]

4.3 Measurement Mechanism

4.3.1 Basic Namespace Register Procedure

The fundamental challenge addressed is distinguishing various containers from the kernel's viewpoint. To achieve this, each container should possess a unique namespace. By parsing the namespace number of the active process during the integrity measurement, the kernel can effectively differentiate between containers. The "namespace register procedure" is devised to enable the kernel to partition ML. In this architecture, the kernel needs to manage multiple double-linked lists to depict the isolated measurement logs from ML.

$$s - ML = \{< value, ns >\}_n = \{< \{measure(ME_{ns})\}, ns >\}_n \tag{4.1}$$

To preserve the integrity of these separated Managed Environments (s-MLs), there is a set of simulated Platform Configuration Registers known as container-based PCRs (cPCRs). Each cPCR is associated with a unique secret and a corresponding namespace number (ns). The secret is a randomly generated value by TPM and is used to conceal the cPCRs of other containers for a specific verifier.

$$cPCR - list = \{cPCR_n\} = \{< value, ns, secret >\}_n \tag{4.2}$$

Upon launching a container, an event is triggered to create a new namespace. Split Hook captures this relevant system call, and the Measurement Agent becomes aware of the event. It then informs the `Namespace Parser`, which proceeds to extract the number associated with the newly created namespace. Subsequently, the cPCR Module generates a new cPCR along with a new s-ML based on this information.

$$cPCR - list := cPCR - list \cup \{AllZero, ns, secret\} \tag{4.3}$$

$$sML := s - MLs \cup \{< \{\}, ns >\} \tag{4.4}$$

4.3.2 Basic Measurement Procedure

The measurement procedure is responsible for measuring and recording MEs. When a measurement event is generated, it activates the IMA hooks. Subsequently, the Measurement Agent is notified, and it measures the respective ME. The results of the namespace number of the current process extracted by the namespace Parser are then passed on to the cPCR module. The cPCR module then proceeds to find the corresponding target cPCR and s-ML.

$$target - cPCR = \{cPCR | cPCR.ns == ns_{num} \cap cPCR \in cPCR - list\} \tag{4.5}$$

$$target - ml = \{s - ML | s - ML.ns == ns_{num} \cup s - ML \in s - MLs\} \tag{4.6}$$

Secondly, the cPCR module extends the current ME into the target cPCR and appends it into the target s-ML

$$target - cPCR.value := HASH(target - cPCR.value, ME.node_{hash}) \tag{4.7}$$

$$target - ml.value := target - ml.value \cup measure(ME) \tag{4.8}$$

The extend operation operates similarly to `PCR_Extend` as defined in TPM standards. The aforementioned steps guarantee that the integrity of each s-ML is safeguarded by its corresponding cPCR. This allows us to verify the authenticity of each s-ML by comparing the related cPCR with the outcome of the simulation.

4.3.3 Bind cPCRs to Hardware-based RoT

Once the target cPCR is extended and the relevant s-ML is appended, the cPCR module proceeds to bind the cPCRs into a physical PCR using the following steps:

- in the current system, a historical value for a specific PCR (`historyPCR`) is maintained in a PCR not used. In the TPM 1.2 based prototype, PCR12 has been designed for this purpose. To allow flexibility a GRUB parameter is offered that enables users to modify this PCR index if needed;
- the system records the digest of all cPCRs. `TempPCR` is initialized as the result of XORing cPCR0.value with cPCR0.secret. Then, for all other cPCRs in the cPCR-list, we apply Equation 7 to extend their values into `tempPCR`. Finally, `tempPCR` will store the digest value representing the combined values of all cPCRs;
- extends the physical PCR12 with the final `tempPCR`.

The secret field is crucial for concealing other containers' cPCRs from a specific verifier. For a user who owns a container with its s-ML extended into cPCR, remote attestation involves sending a nonce to the prover, which serves as protection against replay attacks. The prover then responds with at least the following values to the remote user: `historyPCR`, sendcPCRs, and Sign(AIK, nonce k related PCRs). Here, sendcPCRs are the values extended into PCR12.

In this scenario, the verifier can verify the signature from TPM and then obtain the authentic nonce and PCR12. Subsequently, the verifier validates the genuineness of sendcPCRs by re-calculating a simulated value and extending `historyPCR` with the received digest value. If the calculated result matches the decrypted PCR12, it confirms the trustworthiness of sendcPCRs.

4.4 Extensions

Currently, the Integrity of Containers' dependencies and Integrity of Containers' Boot Time are not recorded separately. The former doesn't create a new namespace, which causes the above-mentioned mechanism to be ineffective in identifying them. As for the latter, a container's images and boot configurations are higher-level concepts in the user space, making it challenging for the kernel to parse and record them. In this section, we propose our solutions to address these issues.

4.4.1 Integrity of Containers' Dependencies

In order to utilize the "namespace register procedure" for these scenarios, a new dedicated program called the bootstrap program is introduced. The primary function of the bootstrap program is to establish a new namespace for an application. Consequently, when launching these dependencies using the bootstrap program, it will activate the "namespace register procedure" accordingly.

In the prototype, the `unshare` [15] command is chosen as the bootstrap program due to Docker-ce's [16] utilization of the `unshare` syscall for allocating new namespaces to containers. The objective is to provide the verifier with the means to verify both the integrity of the bootstrap program and confirm if their containers were genuinely launched using services generated by the bootstrap program. The first issue can be resolved by incorporating the process responsible for creating a new namespace into the relevant s-ML. For example, the `createProcess` action for containers' dependencies pertains to the bootstrap program. As s-MLs are ultimately safeguarded by PCR, the integrity of the bootstrap program is consequently protected as well.

A straightforward approach to tackle the latter issue is to keep a record of the process IDs (PIDs) for the `createProcess` action and its ancestor processes' PIDs. Those PIDs are defined as `createProcess.PIDs`. When a container is launched with authentic dependencies, the PID of the dependencies' `createProcess` can be found within the PIDs of that container. As a result, a verifier can determine the authenticity of the s-ML for the container's dependencies.

4.4.2 Integrity of all Containers' Boot Time

The integrity of containers' boot time encompasses crucial details such as the image, configuration, and parameters required to initialize a container. Since this information resides in the application layer, the kernel cannot directly access it. However, given that the dependencies of the containers have been genuinely recorded and safeguarded by the Root of Trust, we can assign the responsibility of collecting this information to the dependencies. In this way, the dependencies can ensure the integrity of the boot time by gathering and preserving the required data. runc [17] has been modified in a way to do the measurement and extend `nodehash` into another physical PCR, in this case PCR11. A privacy concern arises from the inability to transfer the complete docker boot to the verifier due to its recording of boot information for all containers. Instead, only the `nodehash` for other containers is transferred.

4.5 Attestation Mechanism

It is assumed that an effective user management system exists in a cluster, such as Kubernetes [18], which prevents unauthorized verifiers from initiating the attestation mechanism. Consequently, the identification of verifiers is not taken into consideration. Upon a valid request from a verifier (contains a nonce and the container identifier) the the Attestation Agent in the prover collects:

- the related PCR values and the signature of TPM via performing `TPM_Quote`;
- sendcPCRs and the history value of PCR12;
- s-ML for prover's boot time and the containers' dependencies;
- The s-ML for containers' boot time components targets the entry of the target container, which includes the following information:
  `id, ns, HASH(image), HASH(config), PATH(config), and nodehash`;
- s-ML for the target container's applications.

4.5.1 Workflow of the verifier

- The verifier performs an identity validation of the TPM. Each TPM is assigned a unique endorsement certification by the manufacturer, and this endorsement key serves as a means to identify the TPM. By matching the received PCR values with a correct TPM signature, the verifier can confirm the accuracy of the PCR values. The trustworthiness of the nonce can also be determined, and the verifier is aware of the freshness of this response.
- Relying on the trusted PCR values, the integrity of the s-MLs for both the bootstrapping prover and the target container can be verified. These s-MLs are extended into physical PCRs. To ascertain the authenticity of these s-MLs, a simulation of the `PCR_Extend` operation is performed, and the simulation result is compared with the trusted PCRs.
- The genuineness of the sendcPCRs can be verified using the trusted PCR12. The verifier calculates tempPCR using Equation 7 and the received sendcPCRs. Subsequently, a simulated PCR value is calculated by extending the `historyPCR` with `tempPCR`. If the final sPCR matches PCR12, the verifier obtains trusted sendcPCRs. Conversely, if they do not match, it indicates the occurrence of some attacks.
- The trustworthiness of Container's Applications and Containers' Dependencies can be determined by simulating `PCR_Extend` and comparing the results with cPCRu.value and cPCR0.value, respectively. Additionally, the verifier should verify whether Container's Applications is indeed his container's dependency.
- If all the aforementioned validation procedures yield positive results, all relevant s-MLs are considered trusted. Subsequently, the verifier can validate these s-MLs against his expectations. These expectations are defined to be the same as those in traditional Trusted Computing, and they are collected from software and hardware manufacturers.

4.6 Privacy

The privacy achievements consist of two aspects. Firstly, the information of host applications is not necessarily transmitted to the verifier. In this solution, the s-ML associated with host applications is stored in a separate file. This s-ML is extended into PCR10, following the traditional IMA approach. However, neither this s-ML nor PCR10 is necessary during the remote attestation for containers. Secondly, the boot time information and applications of other containers are not exposed. While all containers' boot time data is recorded into the same s-ML, the information transferred to the verifier only includes the corresponding `nodehash`. As the container id is a random value, the verifier cannot extract meaningful information from the `nodehash`.

A privacy concern is associated with `cPCR.value`. In a standardized container, the boot hash is well-known. If an attacker gains knowledge of `cPCR.value`, they might be able to infer which container is running. In this solution, the `cPCR.values` of other containers are XORed with their corresponding `cPCR.secrets`. The results of this XOR operation are then transmitted to the verifier. Since the `cPCR.secrets` of other containers are not known to the verifier, they cannot reconstruct the original cPCR.values of those containers. This requires a trusted communication between the user and the prover at this moment.

A more stringent privacy requirement is to prevent any verifier from distinguishing whether their container is the only one running on the prover. Denying this knowledge to the verifier thwarts potential attackers from making informed decisions about launching a co-resident attack. In Container-IMA, a malicious verifier can determine the current number of containers running on the prover by counting the size of `sendcPCRs`. To address this issue, obfuscation strategies have been employed. For example, one approach is to introduce a GRUB parameter that configures the prover to operate with a maximum limit of containers. This would limit the verifier's ability to discern the actual number of running containers. The unused cPCRs are populated with meaningless values.

Another related issue is that an attacker can conduct remote attestation multiple times to observe the `historyPCR`. If this value changes, the attacker deduces that another container has executed certain operations. To mitigate this, the prover updates PCR12 when a host application's measurement event occurs. This action creates confusion for the verifier, making it difficult to discern whether the change in `historyPCR` is due to host applications or events within the containers.

4.7 Limitations

The primary objective of this solution is to ensure privacy between containers and prevent excessive disclosure of information about other containers to the verifiers. This advantage is significant, but there is a trade-off since the files created to store the measurements log of the containers are currently deleted when the containers are closed. While this behavior might be deemed acceptable in the current context, it poses a limitation for conducting additional verifications on a container after its closure, as the necessary data will no longer be accessible.

The current trend in the Linux community is to incorporate IMA within its own namespace [19], although existing solutions mainly propose IMA inside the user namespace. Looking ahead, the preferred approach is to adopt a solution that embraces the concept of namespaces for IMA, ensuring more promising prospects for the future.

Overcoming this limitation is the motivation to develop parallel solutions based on the IMA namespace, where a trade-off is done by sacrificing certain privacy features to ensure the persistence of container information and support for namespaces.

Chapter 5

IMA namespacing

Stefan Berger's implementation [20] focuses on creating an IMA namespace within the user namespace. Each user namespace contains a pointer to its corresponding IMA namespace, which includes a measurement list, hash table, and policies. For security purposes, when a user namespace captures an event, it is propagated up to the parent namespace recursively until reaches the host's namespace. This extension ensures that any actions performed within a new namespace are monitored by the host. Without this extension, an attacker could just create a new namespace then close it and leave no trace of their activity.

5.1 Namespace

Namespaces [21] are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources. The feature works by having the same namespace for a set of resources and processes, but those namespaces refer to distinct resources. Resources may exist in multiple spaces. Examples of such resources are process IDs, host-names, user IDs, file names, some names associated with network access, and Inter-process communication.

Namespaces play a crucial role in Linux containers. Initially, a Linux system possesses a single namespace of each type, shared by all processes. However, processes have the ability to create additional namespaces and can also join different namespaces. For each type of namespace, the kernel assigns a symbolic link to every process under the directory `/proc/{pid}/ns/`. These symlinks point to an inode number that remains consistent across all processes within the same namespace. This inode number serves as a unique identifier for each namespace, associated with one of its symlinks.

5.1.1 Namespaces kinds

The functionality of namespaces [21] remains consistent across all types. Each process is linked to a specific namespace, limiting its visibility and access to only the resources associated with that particular namespace and any relevant descendant namespaces. As a result, every process, or group of processes, can possess a distinct perspective of the available resources.

Mount namespace (mnt)

Mount namespaces [22] are responsible for managing mount points. When a new namespace is created, the existing mounts from the current namespace are copied to the new one. However, any mount points created after the creation of the namespace do not propagate across different namespaces.

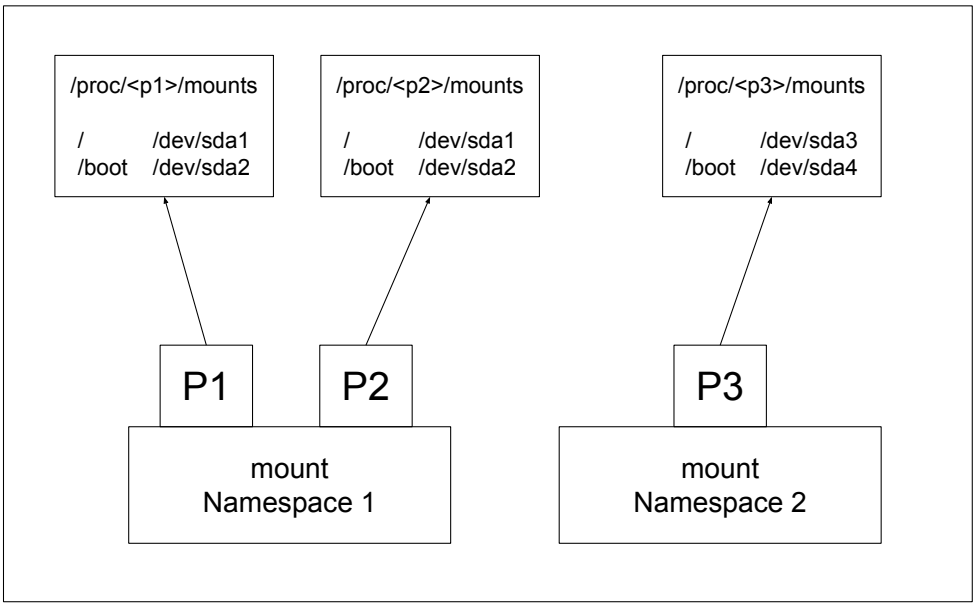

Figure 5.1. Mount namespace schema [22]

Process ID (PID) namespace

PID namespaces grant processes a distinct set of process IDs (PIDs) separate from other namespaces. PID namespaces are organized hierarchically, meaning that when a new process is generated, it obtains a PID for each namespace from its current namespace up to the initial PID namespace. Consequently, the initial PID namespace has visibility over all processes, albeit with distinct PIDs compared to other namespaces. The initial process created within a PID namespace is assigned the PID number 1 and receives similar treatment as the regular init process, including the attachment of orphaned processes within the namespace. Consequently, the termination of this PID 1 process will promptly result in the termination of all processes within its PID namespace, along with any descendants.

Network namespace (net)

Network namespaces [23] serve to create virtualized network stacks. When a network namespace is created, it initially includes only a loopback interface. Each network interface, whether physical or virtual, exists exclusively within a single namespace and can be transferred between namespaces. Each namespace possesses its own isolated set of IP addresses, routing tables, socket lists, connection tracking tables, firewall configurations, and other network-related resources. When a network namespace is destroyed, all virtual interfaces within it are eliminated, and any physical interfaces contained within it are returned to the original network namespace.

Inter-process Communication (IPC) namespace

IPC namespaces [24] provide process isolation for SysV-style inter-process communication. They ensure that processes within different IPC namespaces cannot utilize functions like the SHM family to establish shared memory between them. As a result, each process can use identical identifiers for a shared memory region, creating two separate and distinct memory regions.

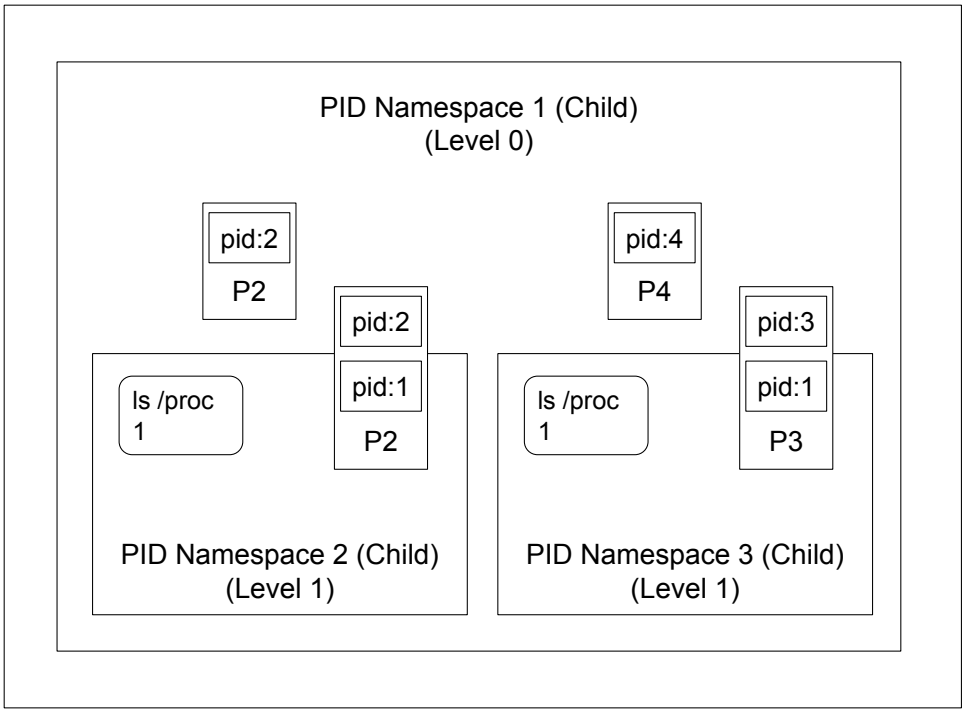

Figure 5.2. PID namespace schema

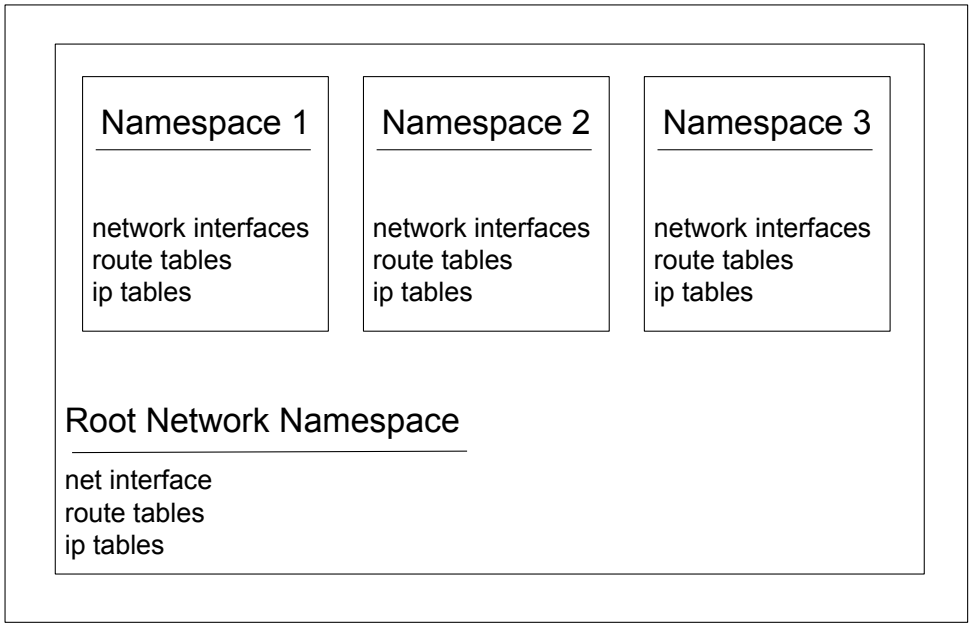

Figure 5.3. Network namespace schema [23]

UTS namespace

UTS (UNIX Time-Sharing) namespaces [25] enable different processes to perceive a single system with distinct host and domain names. When a process creates a new UTS namespace, the hostname and domain of the new namespace are replicated from the corresponding values in the

caller's UTS namespace.

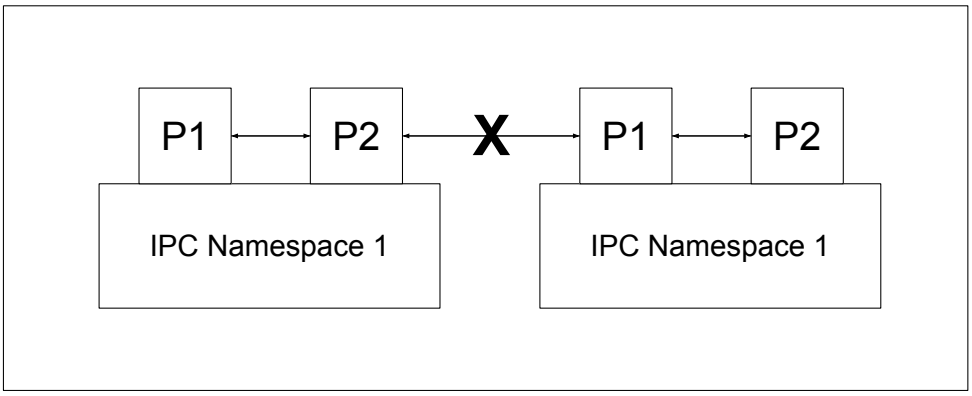

Figure 5.4. IPC namespace schema [24]

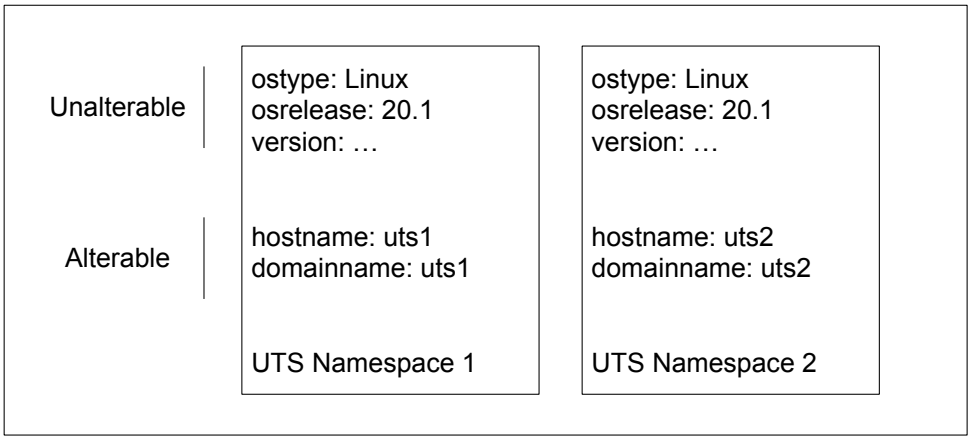

Figure 5.5. UTS namespace schema [25]

User namespace

User namespaces [26] are a feature designed to achieve both privilege isolation and user identification segregation among multiple sets of processes. They allow the creation of containers that appear to have administrative rights without actually granting elevated privileges to user processes. Similar to PID namespaces, user namespaces are hierarchical, with each new user namespace being considered a child of the namespace that created it.

A user namespace includes a mapping table that converts user IDs from the container's perspective to the system's perspective. This enables the root user, for instance, to have user ID 0 within the container while being treated as a user with another ID by the system for ownership checks. A similar table is used for group ID mappings and ownership checks.

To ensure privileged actions are properly isolated, each namespace type is associated with a user namespace based on the active user namespace at the time of creation. A user with administrative privileges in the corresponding user namespace is granted the ability to perform administrative actions within that specific namespace type

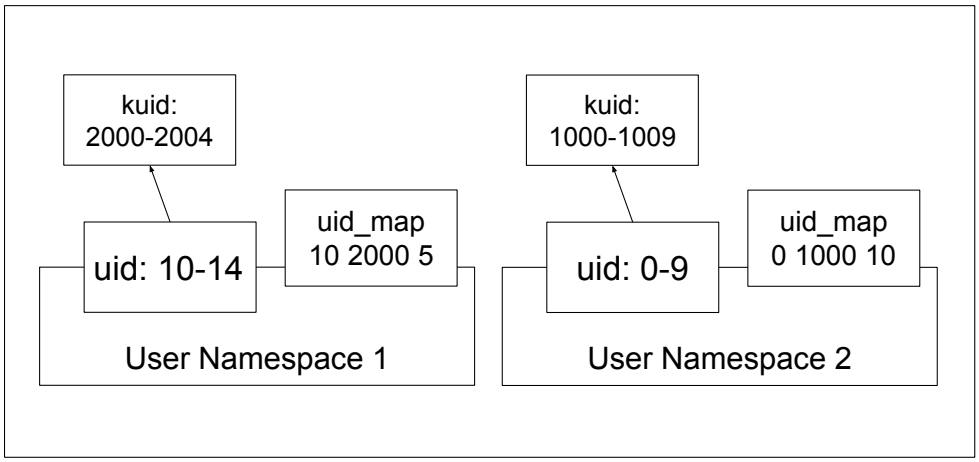

Figure 5.6. User namespace schema

Control group (cgroup) namespace

Control groups, often referred to as cgroups, are a feature embedded within the Linux kernel [27]. They facilitate the arrangement of processes into hierarchical groups, permitting the control and observation of their consumption of diverse resource types. The kernel's cgroup interface is accessed through a pseudo-filesystem known as `cgroupfs`. The grouping functionality is seamlessly integrated into the core cgroup kernel code, while the monitoring and regulation of resources are managed by a suite of subsystems customized for distinct resource categories, including memory, CPU, and others.

Cgroup namespaces [28] provide a virtualized perspective of a process's cgroups. Each cgroup namespace possesses its distinct set of cgroup root directories. These root directories serve as the starting points for the relative locations displayed in the corresponding entries within the `/proc/pid/cgroup` file.

Time namespace

The time namespace [29] provides processes with the ability to perceive distinct system times, similar to the UTS namespace.

5.2 Main Problems Addressed

- **Logging**: within an IMA namespace, files are logged using their absolute paths as they exist within the IMA namespace, which is similar to the pathnames used on the host. However, to enable the logging of IMA measurements outside of an IMA namespace, it is necessary to differentiate them. One approach to achieve this is by creating the absolute path name relative to the IMA/user namespace in which the file is being logged.
- **Securityfs Mount**: to enable support for IMA namespaces, an extension to securityfs is required to allow it to be mounted outside of the initial user namespace. In the context of namespaced users, the pinning logic becomes irrelevant. While in the initial namespace, the securityfs instance and its associated data structures of its users cannot be removed, as previously explained. However, in non-initial securityfs instances, the users associated with the namespace are removed when the last user of the namespace is gone. It is preferable to avoid duplicating the pinning logic or modifying the global securityfs instance to display different information based on the namespace. Implementing either of these options would

result in a messy and hacky solution. Instead, a more preferable approach is to allocate a separate securityfs instance for each namespace. This is similar to how each IPC namespace has its own queue instance, and all entries within it are automatically cleaned up upon unmount or when the last user of the associated namespace is no longer present.

- **Mounting Permissions**: to ensure proper security, it is important to prevent the mounting of a securityfs instance in a user namespace other than the one it belongs to. Additionally, access to files and directories should be restricted to the user namespace to which the securityfs instance belongs or any of its ancestor user namespaces. However, if securityfs is bind-mounted, and thus the init_user_ns is the owning user namespace, access should not be restricted. This is done by controlling the permission at the mount time of a securityfs.

- **Pointer value**: to ensure the validity of the IMA namespace pointer within the user namespace, it is necessary to perform a check. This is due to the possibility of the IMA namespace being either NULL or inactive. The recommended approach is to employ lazy initialization of the IMA namespace, triggered when a user mounts securityfs and writes `"1"` into the `"active"` securityfs file of IMA. This action enables the user namespace to obtain a pointer to an ima namespace. However, prior to utilizing the ima namespace pointer, it is crucial to verify its status using `ns_is_active()`. This check confirms whether the pointer is `NULL` and whether the ima namespace is indeed active.

- **Kernel Memory occupation**: the number of policy rules that a user can set in an IMA namespace that is not the host's one (`non-init_ima_ns`) should be limited to a hardcoded value of `"1024"`. This restriction is in place to control the amount of kernel memory utilized by IMA's policy. The purpose of this limitation is to prevent any user from creating an IMA namespace and potentially wasting kernel memory. If a user attempts to exceed this limit by setting too many additional rules, those extra rules should be disregarded or ignored.

- **Store namespace iint data**: a new feature can be introduced by including an RB-tree in the IMA namespace structure, specifically designed to store a namespaced version of `iint-flags` within the `ns_status` struct. This RB-tree operates similarly to the `integrity_iint_cache`, where both the `iint` and `ns_status` can be retrieved using the inode pointer value. The process of looking up, allocating, and inserting values into this RB-tree follows a comparable code structure as well. There are two cases for freeing, when the `iint` is freed and when the ima namespace is freed.

- **clean of unused iinit**: an `iint` is considered unused when it satisfies the following two conditions:

  - its `ns_status` list is empty, indicating that no IMA namespace currently stores any auditing-related state within it;
  - the `iint`'s flags do not include any of the flags `IMA_MEASURE`, `IMA_APPRAISE`, or `IMA_HASH`, which are still stored by the host.

  For these cases, an ns_status list is unnecessary. However, a `ns_status` list is required for `IMA_AUDIT`. To process the list of garbage-collected `ns_status`, it is recommended to perform the processing outside the lock of the `ns_status` tree and its related lock-group. During this processing, any `iint` that was previously identified as unused while traversing the list should be freed.

5.3 Structure

The most important fields of the IMA namespace struct are:

- `list_head` is the generic double-linked list implementation, a standard way for linked list creation in the Linux kernel;

- `ima_htable` is a standard fixed-length hash table that uses the template hash as key;

- `ima_measurements`: contains the list of measurements related to this IMA namespace.

The `__randomize_layout` keyword at the end of the struct declaration is a GCC plugin implemented in the Linux kernel. The purpose is to incorporate it into the structure definitions to enable the compiler to randomize the field order. This technique is employed for security reasons to mitigate attacks that rely on knowledge of the structure layout.

```
struct ima_namespace {
    ...
    /* List of active rules */
    struct list_head *ima_policy_rules;
    ...
    /* Hash table for the ima_measurements */
    struct ima_h_table ima_htable;
    ...
    /* list of all measurements */
    struct list_head ima_measurements;
    ...
} __randomize_layout;
```

Listing 5.1. IMA namespace struct

The event propagation to the parent namespace occurs in `ima_main` within the process measurement function. This leverages the insertion of the IMA namespace into the user namespace. By obtaining the IMA namespace from the user, we can assess the namespace activity, handle the event, and continue looping up to the parent namespace until we reach the host.

```
while (user_ns) {
    ns = ima_ns_from_user_ns(user_ns);
    if (ns_is_active(ns)) {
        int rc;
        ...
        rc = __process_measurement(ns, file, cred, secid, buf,
               size, mask, func);
        ...
    }
    user_ns = user_ns->parent;
}
```

Listing 5.2. Parent extension of the measure process

To activate the IMA namespace is needed to print the value to 1 in the file `sys/kernel/security/integrity/ima/active`. This file is part of the securityfs, so any write operation to it is regulated. Once the write operation is completed, the function `ima_init_namespace` is invoked to allocate and initialize the namespace. This is the function of the filesystem that regulates the write in the file. So the concept of the IMA filesystem is extended with the active file and other small enhancements.

```
static ssize_t ima_write_active(struct file *filp, ...) {
    ...
    if (ns_is_active(ns))
        return -EBUSY;

    /* accepting '1\n' and '1\0' and no partial writes */
    if (count >= 3 || *ppos != 0)
        return -EINVAL;

    kbuf = memdup_user_nul(buf, count);
    if (IS_ERR(kbuf))
        return PTR_ERR(kbuf);
    err = kstrtouint(kbuf, 10, &active);
```

```
    kfree(kbuf);
    if (err)
        return err;
    if (active != 1)
        return -EINVAL;
    err = ima_init_namespace(ns);
    if (err)
        return -EINVAL;
    return count;
}
```

Listing 5.3. Function to handle the activation of the IMA namespace

In the file ima_policy.c, the function ima_parse_rule demonstrates that all policies, except for the audit policy, are ignored. The switch case statement only considers the audit policy. Only the IMA namespace of the host can incorporate all policies. This restriction is due to the missing implementation of the extension in the parent chain of the other rules. For this reason, instead of generating an error, the insertion is simply ignored.

```
/* IMA namespace only accepts AUDIT rules */
if (ns != &init_ima_ns && result == 0) {
    switch (entry->action) {
        case MEASURE:
        case APPRAISE:
        case HASH:
            result = -EINVAL;
            goto err_audit;
        case AUDIT:
            if (!may_set_audit_rule_in_ns(current_uid(), user_ns)) {
                result = -EPERM;
                goto err_audit;
            }
    }
}
```

5.4 Testing

To perform tests we use `BusyBox` [30] which is a popular open source project offering a compact implementation of approximately 400 common UNIX/Linux commands. Its minimal image size, fitting under 1 MB, has contributed to its popularity in embedded systems, IoT, and cloud computing. Could be installed using:

```
$ sudo apt install busybox
```

5.4.1 Unshare command

The `unshare` [15] command creates new namespaces, specified by the command-line options below, and then runs the specified program. If no program is provided, it defaults to `{SHELL}` (/bin/sh).

By default, a fresh namespace remains active only while it contains member processes. Nevertheless, it can attain persistence by associating `/proc/{PID}/ns/type` files with a location within a filesystem. This enables access to the enduring namespace even after the program concludes, except in the case of PID namespaces, which necessitate a functioning init process. To eliminate the association created by bind mounting and revert a namespace to a non-persistent state, the umount command is employed.

Unshare can create different types of namespace, the main ones:

- **mount namespace**: except for explicitly marked shared filesystems, mounting and unmounting filesystems have no impact on the rest of the system;
- **PID namespace**: children will have a distinct set of PID-to-process mappings from their parent;
- **user namespace**: the process will have a distinct set of UIDs, GIDs, and different capabilities.

The used options are (for more information [15]):

- `-U, --user`: create a new user namespace;
- `-r, --map-root-user`: execute the program once the current user and group IDs are mapped to the superuser UID and GID in the newly created user namespace. This enables the acquisition of necessary capabilities to manage different aspects of the newly created namespaces, even when running without privileges;
- `-p, --pid`: create a new PID namespace;
- `-f, --fork`: instead of executing the specified program directly, create a child process of unshare to fork it. This approach is beneficial when creating a new PID namespace;
- `-R, --root=dir`: run the command with the `root` directory set to `dir`;
- `--mount-proc`: just before running the program, mount the proc filesystem, the default mount point is `/proc`.

5.4.2 Run the test

The test aims to verify that the event is audited and propagated to the parent namespace. The test must be executed with root privileges. To capture the audit performed by the child namespace, you need to add a policy on the host. This policy should be designed to capture the relevant audit events from the child namespace, as it is not extended natively in this solution.

```
$ echo -e "measure func=BPRM_CHECK mask=MAY_EXEC uid=1000\n" \
  "audit func=BPRM_CHECK mask=MAY_EXEC uid=1000\n" \
  > /sys/kernel/security/ima/policy
```

A simulated filesystem is generated where the Busybox files are placed, which are copied from the host's Bin directory. Two duplicates are created, and one of them is modified to alter the hash value, resulting in two distinct measurements. By executing the unshare [15] command, a new namespace is created, and a shell is opened within it. To enable the IMA namespace, the value `"1"` must be written in the `"active"` file, after which the policy can be configured. Here are reported the commands used:

```
$ mkdir -p rootfs/{bin,mnt,proc}
$ cp /sbin/busybox rootfs/bin
$ cp /sbin/busybox rootfs/bin/busybox2
$ echo >> rootfs/bin/busybox2
$ PATH=/bin unshare --user --map-root-user --mount-proc \
  --pid --fork --root rootfs busybox sh
$ busybox mount -t securityfs /mnt /mnt
$ busybox echo 1 > /mnt/ima/active
$ busybox echo 'audit func=BPRM_CHECK mask=MAY_EXEC' > /mnt/ima/policy
$ busybox2 cat /mnt/ima/policy
```

The output should be a message similar to this:

```
audit: type=1805 audit(1684408045.330:2): file="/root/rootfs/bin/busybox2"
  hash="sha1:1f66ff4ef78df9ad7cd44f4e2359ef3a9fe9f630" ppid=123 pid=131
  auid=0 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=tty1
  ses=1 comm="busybox" exe="/bin/busybox" subj=unconfined
  audit func=BPRM_CHECK mask=MAY_EXEC
```

That shows a correct audit message, explaining the type, path of the file, the hash calculated, the group and user information, and other values.

5.5 Strong points and Limitations

- only the policy for audit is accepted, the insertion of other policies is refused, and the other rules are not implemented;
- the IMA namespace has no identifier, so it's hard to know where the measure comes from;
- given that no measure policy is supported there is no way to check the measure effectuated;
- no specifications about the infrastructure needed to attest the values.

Chapter 6

Keylime

Keylime originated within MIT's "Lincoln Laboratory" security research team and was introduced to the academic community in December 2016 through the publication of the whitepaper titled "Bootstrapping and Maintaining Trust in the Cloud." [31]. At present, it stands as a project hosted by the "Cloud Native Computing Foundation" (CNCF), furnishing an open-source resolution for initiating hardware-rooted cryptographic identities for cloud nodes, along with conducting regular attestation to monitor the system integrity of these nodes.

6.1 Introduction

The concept behind Keylime originates from recognizing the increasing popularity of "Infrastructure as a Service" (IaaS) offerings among businesses. "Infrastructure as a Service" (IaaS) stands as a cloud computing service model in which essential computing resources (such as processing power, storage, and networks) are allocated to customers for the purpose of deploying and executing various software, encompassing operating systems and applications. These resources within the IaaS framework are denoted as cloud nodes, encompassing physical hardware, virtual machines, or containers. Within this service model, users of the cloud, known as tenants, don't oversee the management of the underlying cloud infrastructure. Instead, they maintain control over operating systems, storage, and the applications they deploy. Commonly, customers possess the capability to independently provision this infrastructure by utilizing a web-based graphical user interface, which serves as an IT operations management console for the entire environment. Nowadays, IaaS enjoys substantial popularity within the corporate landscape, primarily due to its capacity to provide cutting-edge technologies while delivering time and cost savings. This enables companies to:

- evade the initial costs associated with establishing and overseeing a data center;
- quick implementation of innovations;
- concentrate on the core business of the company;
- gain entry to applications and data from any location and at any moment.

Nevertheless, IaaS cloud service providers presently do not furnish all the necessary elements to create a secure setting for hosting sensitive data and critical business applications. Tenants possess restricted capability to authenticate the foundational platform as they introduce their applications to the cloud and to ensure the platform's trustworthiness throughout the entirety of their computational processes.

A potential remedy for establishing initial trust and identifying alterations in the system's condition is offered by the TPM. However, its adoption in IaaS cloud environments has been limited due to the intricacies of its standards, challenges in implementation, and performance

limitations. Considering that the TPM is a physical device while the majority of IaaS services operate on virtualization. Collection of favorable attributes that an IaaS system based on trusted computing should possess:

- **Secure Bootstrapping**: the system should enable a tenant to securely introduce an initial root secret into each of their cloud nodes. Subsequently, this initial secret can be utilized by the tenant to establish a sequence of additional secrets, thereby facilitating the implementation of enhanced security services.
- **Secure Layering**: the system should empower a tenant to achieve secure bootstrapping and integrity monitoring, even within virtual machines, by leveraging a TPM integrated into the provider's infrastructure. This necessitates collaboration with the provider while ensuring that the provider is granted the minimum necessary privilege.
- **Scalability**: the system should be capable of scaling to accommodate secure bootstrapping and integrity monitoring for a multitude of cloud nodes, as IaaS resources can be dynamically created and removed.
- **System Integrity Monitoring**: the system should provide the tenant with the capability to oversee the integrity status of cloud nodes and swiftly identify any discrepancies, responding within a one-second timeframe.
- **Compatibility**: the system should grant the tenant the ability to utilize hardware-rooted cryptographic keys within the software, thereby enhancing the security of the services they are currently employing.

Keylime has been introduced as a resolution to address security challenges inherent in cloud computing environments. It possesses the capacity to furnish tenants with a hardware-rooted foundation of trust, thereby enabling the verification of the reliability of both the IaaS infrastructure and the tenant's own systems operating within that environment.

Keylime [31] represents a TPM-driven, immensely adaptable solution for remote boot attestation and runtime integrity measurement. By harnessing a hardware-based cryptographic root of trust, Keylime empowers cloud users to efficiently oversee distant nodes.

The Keylime architecture conforms to the illustrated diagram, which illustrates the primary components and their principal interactions with each other.

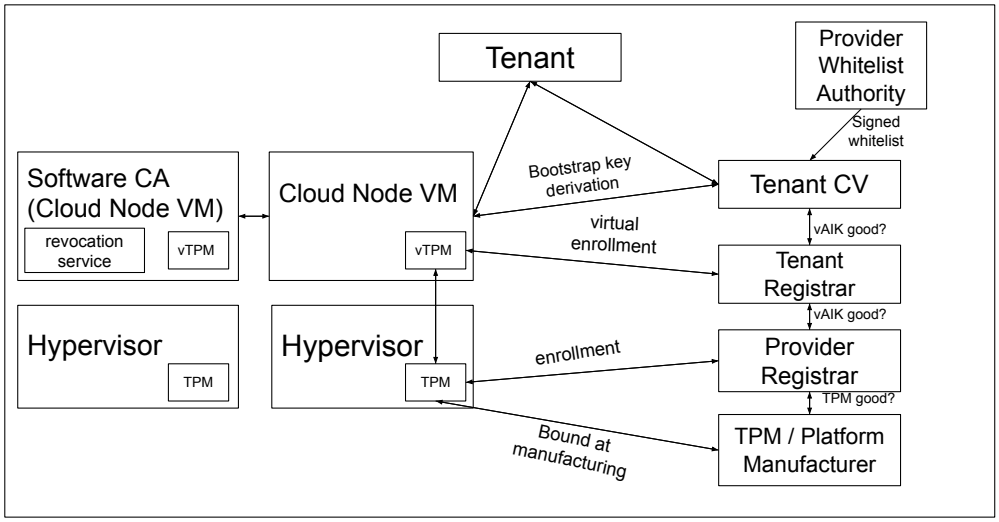

Figure 6.1. Keylime main schema [32]

6.2 Main components

Keylime consists mainly of an agent, two server controllers, and a command line tool. Each component is developed independently and communicates with each other using APIs. Presently, it exists as a project hosted by the "Cloud Native Computing Foundation" (CNCF) and offers an open-source solution for establishing hardware-rooted cryptographic identities for cloud nodes, as well as for monitoring the system integrity of those nodes through periodic attestation.

- **Agent**: is a service that runs on the operating system that should be attested. It interacts with the TPM to enroll the Attestation Key, generate quotes, and gather essential data to facilitate state attestation. The agent can monitor revocation events dispatched by the verifier in case of failed agent attestation. This feature proves beneficial in environments where attested systems engage in direct communication, necessitating mutual trust among them. In such scenarios, a revocation message could trigger adjustments to local policies, preventing the compromised system from accessing any resources belonging to other systems.
- **Registrar**: the agent undergoes registration with the registrar, which oversees the entire enrollment procedure. This involves obtaining a Universally Unique Identifier (UUID) for the agent, gathering the EKpub, EK certificate, and AKpub from the agent, and subsequently validating that the AK is associated with the EK. After successful registration in the registrar, the agent becomes eligible for attestation enrollment. The tenant can utilize the EK certificate to assess the reliability and trustworthiness of the TPM.
- **Verifier**: is responsible for executing the agent's attestation process and issuing revocation messages whenever an agent deviates from the trusted state. Upon the agent's registration for attestation, either through the tenant or directly via the API, the verifier consistently retrieves the necessary attestation data from the agent. This data may encompass a quote covering the PCRs, PCR values, NK public key, IMA log, and UEFI event log.
- **Tenant**: serves as a comprehensive platform for agent management. It facilitates tasks such as adding or removing agents from attestation, validating the EK certificate against a certificate store, and retrieving the status of a particular agent. Additionally, the tenant equips users with essential tools for handling the payload mechanism and conducting revocation actions.

6.3 Main functionalities

- **User Selected PCR Monitoring**: by leveraging the tpm_policy feature in Keylime, it becomes feasible to monitor a remote machine for changes in any specified PCR.
- **Measured boot**: Currently, the UEFI firmware has made the event log accessible through an ACPI table, and it is now utilizing this table to expose the boot event log through securityfs. Typically, this log is accessible at the path `/sys/kernel/security/tpm0/binary_bios_measurements`.

  By combining this feature with secure boot, the designated PCR set can be fully populated, incorporating measurements of all components, up to the kernel and `initrd`. The `initrd` [33], or initial RAM disk, enables the boot loader to load a RAM disk. This RAM disk can be utilized as the primary file system, allowing programs to run from it. Later on, a new file system can be mounted from a different device, while the original root file system (from `initrd`) is relocated to a directory and can be unmounted as needed. The primary purpose of `initrd` is to facilitate a two-phase system startup. Initially, the kernel initializes with a basic set of built-in drivers. Subsequently, additional modules are loaded from `initrd`. In addition to the boot log data sources mentioned above, users can utilize `tpm2-tools` to consume the contents of such logs and thereby reconstruct the contents of PCRs [0-9] (and potentially PCRs [11-14]).

  Keylime can leverage this newfound capability with great flexibility. It allows the Keylime operator to specify a "measured boot reference state" or `mb_refstate` for brevity. This

operator-provided data is then utilized by the `keylime_verifier` in a manner akin to the "IMA policy." The `keylime_verifier` compares the information received from the `keylime_agent` against this reference state to ensure integrity and security.

- **Runtime Integrity Monitoring**: Keylime's runtime integrity monitoring necessitates the establishment of Linux IMA. It also offers support for verifying IMA file signatures, a capability that aids in detecting modifications on immutable files. Moreover, it can complement or potentially replace the allowlist of hashes in the runtime policy if all pertinent executables and libraries are signed.

  In the event that the IMA measurement log contains invalid signatures, a system reboot is required to initiate a clean log. This ensures that the Keylime Verifier can effectively verify the system's integrity.

- **Secure Payloads**: provide a means to securely provision encrypted data to an enrolled node. This encrypted data serves to deliver essential secrets required by the node, such as keys, passwords, certificate roots of trust, and other sensitive information. Secure payloads are designed to address any aspect necessitating robust confidentiality and integrity, offering an effective way to bootstrap the system securely. The payload is encrypted and transmitted through the Keylime Tenant CLI (or REST API) to the Keylime Agent. Along with the encrypted payload, the Agent also sends a key share called the u_key or user key, which is essential for decrypting the payload. Keylime provides two modes for sending secure payloads:

  - **single file encryption**: when you specify a file to the `keylime_tenant` application using the `-f` option, the Tenant encrypts the file using the bootstrap key and securely delivers it to the Agent;
  - **package mode**: automates numerous common actions that tenants frequently perform while provisioning their Agents. Firstly, Keylime has the capability to create an X509 certificate authority, and it also natively supports certificate revocation.

Chapter 7

Proposed Solution

7.1 Problems addressed

To establish a functional IMA environment based on Stefan Berger's solution [20], the initial implementation requires the measure rule function. Recursive extension of all measures up to the host is necessary to ensure integrity checks possible even if the namespace has been closed. Without a recursive extension of measures up to the host, an attacker could potentially create a namespace, carry out actions within it, and close it without leaving any evidence of their actions. Each of the suggested solutions relies on establishing a novel and distinct identifier for the IMA namespace. This identifier plays a crucial role in identifying the origin of a measure.

7.2 First Approach

7.2.1 Idea

The concept involves expanding the application of parent extension, as seen in Stefan Berger's IMA namespaces implementation, to encompass a measurement rule. This comprehensive approach ensures the solution operates effectively and facilitates attestation. The initial solution proposed aims to enable container attestation even when closed. To achieve this, a new IMA template is introduced, which incorporates two additional fields. One field pertains to the identifier of the IMA namespace, while the other field represents a counter indicating the number of extensions made on the PCR (Platform Configuration Register) using that particular measure. This counter value allows for the reconstruction of the measurement list entry of a child namespace, even if it had been closed. In this way, all the measures are reported in the host's list. This gives the host higher control on the executed programs on the namespaces but to verify a namespace we need to verify the whole host list.

Policies

An essential aspect of the solution involves how policies are handled. For the implementation of IMA namespaces, each namespace will possess its unique IMA policy. These policies can be inserted during namespace creation or dynamically enabled. Setting the namespace policy during creation requires kernel configuration. Even if a measure event originated from a distinct namespace that does not match any policy, it will still be measured and extended to the parent. This forced measurement ensures the completeness of the measurement chain, facilitating verification processes.

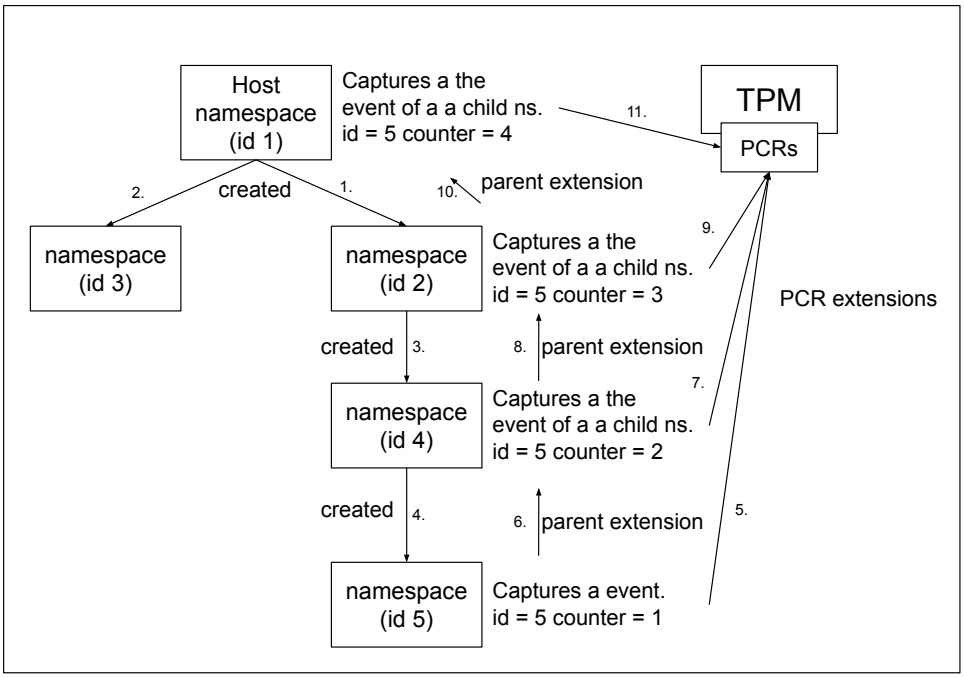

Figure 7.1. IMA namespace solution with counter schema

7.2.2 Implementation

IMA namespace identifier

Insertion of a unique identifier for the namespace that can be randomly generated or incrementally generated starting from one.

```
struct ima_namespace {
    int id;
    ...
} __randomize_layout;
```

Listing 7.1. Identifier of the IMA namespace

Some functions permit the retrieval of these values given the user namespace, this is used to perform the mapping between the container identifier and the IMA namespace identifier as we will see in the next chapter.

```
static inline u32 ima_ns_id_from_user_ns(struct user_namespace *user_ns) {
    struct ima_namespace *ima_ns = ima_ns_from_user_ns(user_ns);
    return ima_ns->id;
}
```

Listing 7.2. Function to retrive the IMA namespace identifier from the user namespace

Template

The implementation of the template now includes an ad-hoc function to manage the insertion of two new custom fields: the counter and the IMA namespace identifier. These functions play a crucial role in handling the generation of template data within a template entry. Template data, being variable and dependent on the specific template, is now effectively handled using this approach.

```
static struct ima_template_desc builtin_templates[] = {
    ...
    {.name = "ima-nsid-cnt", .fmt = "d-ng|n-ng|id|num_mes"},
};

static const struct ima_template_field supported_fields[] = {
    ...
    {.field_id = "d-ng", .field_init = ima_eventdigest_ng_init,
     .field_show = ima_show_template_digest_ng},
    {.field_id = "n-ng", .field_init = ima_eventname_ng_init,
     .field_show = ima_show_template_string},
    {.field_id = "id",
     .field_init = ima_id_init,
     .field_show = ima_show_template_id},
    {.field_id = "num_mes",
     .field_init = ima_num_mes_init,
     .field_show = ima_show_template_id},
};
```

Listing 7.3. Modifications to the IMA template

Atomic parent extension

In the standard IMA implementation, atomicity is assured solely between the insertion in a list and the extension of the PCR. To prevent a scenario where one namespace's PCR extension interrupts another namespace's parent extension, a lock becomes necessary. Without this lock, if one namespace extends to the parent while another namespace simultaneously performs a PCR extension, the order of PCR extensions becomes indeterminate. As a result, it becomes impractical to verify the list with the PCR value accurately.

To reduce waiting time during PCR extension, a list is introduced and implemented as a circular buffer following the "First-In-First-Out" (FIFO) schema. This list keeps track of namespace IDs that wish to perform extensions, ensuring they wait their turn accordingly. When a namespace needs to execute PCR extensions, it locks a mutex on the vector, adds its identifier to the first available slot, and then releases the lock. The process then waits until a pointer points to the element in the vector corresponding to its identifier. All PCR extensions are executed in sequence, and subsequently, the pointer of the vector is incremented to continue the orderly process.

```
...
if(starting_ima_ns_id == ns->id) {
    mutex_lock(&vett_queue_mutex);
    vett_queue[next_empty_slot] = ns->id;
    next_empty_slot = (next_empty_slot + 1)% MAX_VETT_QUEUE_LEN;
    mutex_unlock(&vett_queue_mutex);
}

while(vett_queue[actual_id] != starting_ima_ns_id);
```

Listing 7.4. Locking mechanism during parent extension operation

Process measurement

Here is shown the code to perform the parent extension. The starting IMA namespace is taken and saved in a way that every parent knows from which namespace the measure comes. The parent namespace is obligated to save the value in its list and perform the PCR extension.

```
...
if(user_ns != NULL) {
```

```
    starting_ima_ns = ima_ns_from_user_ns(user_ns);
    while(starting_ima_ns == NULL) {
        starting_ima_ns = ima_ns_from_user_ns(father->parent);
        father = father->parent;
    }
    starting_ima_ns_id = starting_ima_ns->id;
}

while (user_ns) {
    ns = ima_ns_from_user_ns(user_ns);
    if (ns_is_active(ns)) {
        num_measurements++;
        rc = __process_measurement(ns, file, cred, secid, buf,
                    size, mask, func, num_measurements, starting_ima_ns_id);
        ...
    }
    user_ns = user_ns->parent;
}
```

Listing 7.5. Modified parent extension operation

Measurement list

Here is an example of what the measurement list looks like. The two new fields of the template are inserted to indicate the IMA namespace identifier of the namespace that triggered the event and the counter. The first measure is generated by the host namespace, so it has the identifier `"1"` and the counter `"1"` because the measure has not been extended in other namespaces. The second measure comes from a direct child namespace, with identifier two, and the counter is two because it is extended into the host namespace, and so on.

```
10 f4...09 ima-nsid-cnt sha1:59...d8 /usr/bin/cat 1 1
10 e2...11 ima-nsid-cnt sha1:24...1a /usr/bin/echo 2 2
10 a4...44 ima-nsid-cnt sha1:39...c4 /usr/bin/ls 5 3
```

Listing 7.6. Example of entries in the list for this implementation

7.2.3 Verification

One of the key advantages of this solution is its ability to verify even closed namespaces. It allows for the reconstruction of an entry originating from a child namespace by modifying the counter value and subsequently recalculating the template hash. The template hash, which is a hash based on the template values, is then used to extend the PCR. So no namespaces' lists are needed to perform integrity checks, all the information is contained in the host's list.

During the simulation of PCR extension for the host's list, there is a possibility of encountering a value from another namespace. This value is necessary to generate the values present in the child namespaces' lists. The values in the child namespaces' lists will be identical to the ones in the host's list, but the only difference is the counter value. To reconstruct them, new entries are created with a counter that starts at one and increments until it reaches the counter value present in the host list.

Here's a brief example of the steps needed to simulate an entry of a child namespace starting from the host's entry using a Python script:

```
for x in range(1,number_of_extensions):
    value_to_hash = bytearray(algo_used.encode('utf-8')) + digest_file

    path = tokens[-3] + '\x00'
    value_to_hash = value_to_hash + bytearray(path.encode('utf-8'))
```

```
value_to_hash = value_to_hash + bytearray((tokens[-2] +
                str(x)).encode('utf-8'))
value_hashed = Hash.compute_hash(hash_alg, value_to_hash)
runninghash = Hash.compute_hash(hash_alg, runninghash + value_hashed)
```

Listing 7.7. Verification logic to reconstruct entries in child namespaces

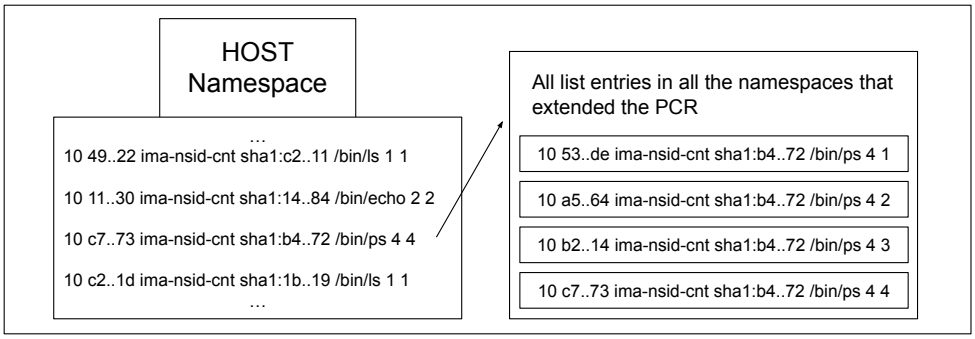

Figure 7.2. Integrity checks on the host's list

7.2.4 Strong points and weaknesses

This solution offers several strengths. Firstly, it allows independent verification of a namespace (or container), whether it is open or closed. Additionally, it provides direct control over the execution within the namespace, simplifying the attestation process.

However, there are certain weaknesses to consider. The main drawback is the lack of privacy between namespaces. When attesting a namespace, the entire host list needs to be attested, which can compromise the privacy of individual namespaces. Guaranteeing the integrity of a specific namespace's list requires attesting the entire host's list, including values from other namespaces. Consequently, when requesting the attestation of a container, the entire list is disclosed, including values originating from other namespaces.

7.3 Second approach

7.3.1 Idea

The second approach revolves around monitoring the creation and destruction of namespaces, with the objective of eliminating the counter while maintaining an alternative method to track the number of PCR extensions for an event. When a namespace is created, an entry is added to the namespace's measurements list (Figure 7.3). This entry has a unique template containing three fields: one to indicate whether it's a creation or destruction event, another to identify the parent IMA namespace of the target namespace, and the last one to indicate the IMA namespace identifier of the namespace that was created or destructed. This entry is propagated up the hierarchy, being extended to all parent lists, notifying each host namespace about the creation of this particular namespace.

The measures in the list now include an additional field indicating the identifier of the IMA namespace. This proves useful in identifying the source of a given measure. With this approach, we no longer require a counter since the host, equipped with information about the namespace creation (Figure 7.4), can reconstruct a tree of creations and determine how many times the measure has been re-extended.

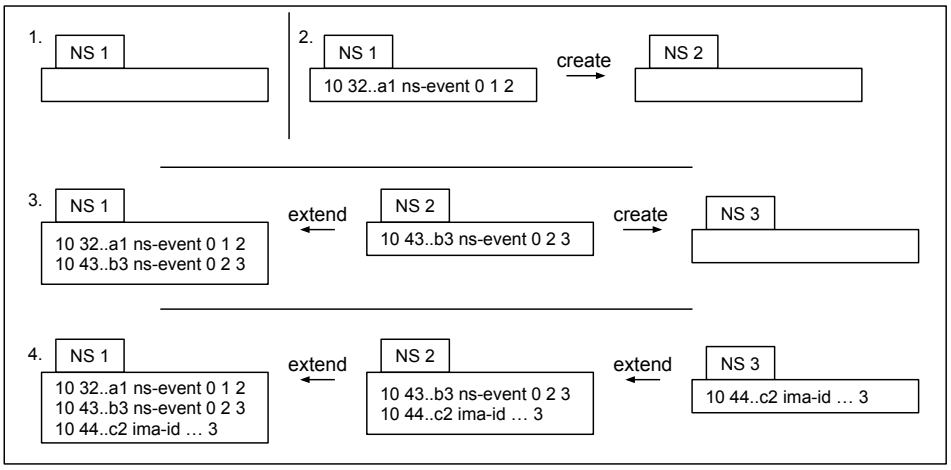

Figure 7.3. Events of creation new namespace and register of events

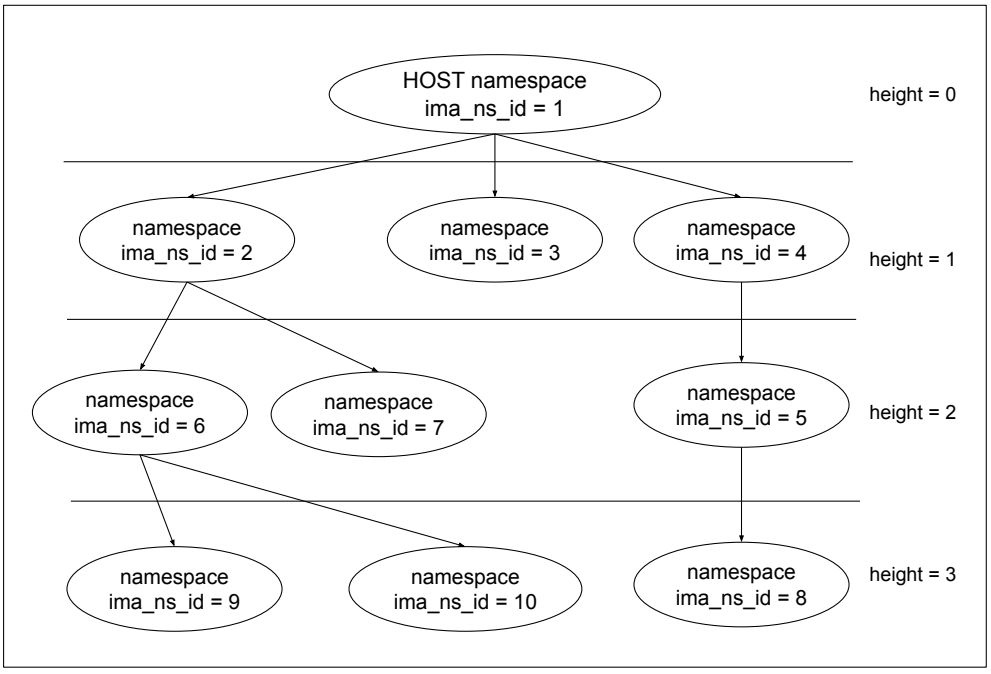

Figure 7.4. Namespace creation tree example with heights

7.3.2 Implementation

Template

Insertion of the three new custom fields, to handle the insertion in the list for the values regarding the type of the event registered, the parent of the target of the event, and the namespace that has been created or closed.

```
static const struct ima_template_field supported_fields[] = {
    ...
    {.field_id = "imaidcreator",
     .field_init = ima_id_creator_init,
```

```
     .field_show = ima_show_template_string},
    {.field_id = "imaidcreated",
     .field_init = ima_id_created_init,
     .field_show = ima_show_template_string},
    {.field_id = "imaeveninfo",
     .field_init = ima_event_info_init,
     .field_show = ima_show_template_string},
};

static struct ima_template_desc builtin_templates[] = {
    ...
    {.name = "ns-event", .fmt = "imaeveninfo|imaidcreator|imaidcreated"},
};
```

Listing 7.8. New IMA template for event registration

Creation of a namespace

Insertion of a function to manage the creation of a template entry and populate it with the required values during the namespace creation process. The current implementation of IMA namespaces initializes this process using the function `write_active`. This function also takes care of the parent extension of the measure. It is crucial to maintain a record of all created namespaces for security and verification purposes.

```
void ima_ns_event(struct ima_namespace *ns_creator, struct ima_namespace
       *ns_created, ...) {
    ...
    event_data->event_info = event_info;
    event_data->ima_ns_id = ns_creator->id;
    event_data->ima_ns_id2 = ns_created->id;
    template_desc->num_fields = template_num_fields;
    template_desc->fields = fields_const;
    ...
    result = ima_alloc_init_template(event_data, &entry, template_desc);
    result = ima_store_template(ns_to_extend, entry,
           violation, NULL, NULL, PCR);
    ...
    return;
}
```

Listing 7.9. Function to extend the registration of a namespace creation or closure event

Closure of a namespace

Also, the event of closure is registered, to keep a trace of the status of a namespace in an easy way. The function that handles the closure of a namespace is `free_ima_ns`, it also handles the extension in all the parent chains up to the host.

```
void free_ima_ns(struct user_namespace *user_ns) {
    ...
    ima_free_ima_ns(ns);
    while(to_extend) {
        ima_ns_to_extend = ima_ns_from_user_ns(to_extend);
        if(ima_ns_to_extend != NULL && ns_is_active(ima_ns_to_extend))
        ima_ns_event(ima_ns_parent_of_destroyed, ima_ns_to_destroy,
                  1, ima_ns_to_extend);
        to_extend = to_extend->parent;
    }
```

```
    ima_free_ima_ns(ima_ns_to_destroy);
    user_ns->ima_ns = NULL;
}
```

Listing 7.10. Registration of namespace closure

Measurement list

The Measurement list will have different templates inside, is useful to know the identifier of the namespace that started the measure so a different template is inserted with just the IMA namespace identifier. In the example is possible to see the creation of the namespace with an IMA namespace identifier equal to 2 and at the end its destruction, in the middle there are two measurements, one of the host and the other one of the newly created namespace, the one with id two.

```
10 54...12 ns-event 0 1 2
10 3d...39 ima-id sha1:cc...dd /usr/bin/cat 1
10 32...1a ima-id sha1:08...36 /usr/bin/echo 2
10 a2...14 ns-event 1 1 2
```

Listing 7.11. Example of measurement list

7.3.3 Verification

To ensure the integrity of the list, it is essential to reconstruct the namespace creation tree. As the list is scrolled through, whenever a namespace creation event is encountered, a node is created and inserted as a child of the creator of the namespace. On the other hand, when a measure from another namespace is encountered, it becomes necessary to determine the number of times this measure has been re-extended. This number corresponds to the height 7.4 of the tree plus one, which is calculated starting from the node representing the namespace. Since the entry does not change in different lists is only needed to perform the re-extension with the same entry. The code provided demonstrates how the insertion in the tree is handled and how an entry from another namespace is managed.

```
if tokens[2] == 'ns-event':
    if str.isdigit(id_parent) and str.isdigit(id_ns):
        parent = int(id_parent)
        child = int(id_ns)
    else:
        print("wrong ima log structure")
        break
    if ns_event_info == 0:
        # creation of a namespace, insertion in the tree
        tree.add_child_between(tree.find_in_childs(parent),
                    tree.find_in_childs(child))
    elif ns_event_info == 1:
        # closure of the namespace
        tree.close_ns(child)
...
# number of time the PCR extension have to be simulated
number_of_extensions = tree.node_heigh(tree.find_in_childs(tokens[5]))
...
```

Listing 7.12. Namespace creation tree implementation

7.3.4 Strong points and weaknesses

The weaknesses of the initial solution still persist. Privacy between containers remains an issue, as attestation must be carried out collectively. The host's list remains essential for conducting any integrity checks. However, a significant advantage of this solution lies in its capability to reconstruct the tree of namespace creation. This grants the host greater control over the creation of namespaces. Nevertheless, it is crucial to address the privacy concerns and devise methods to perform attestation individually for enhanced container security.

7.4 Third solution

7.4.1 Idea

The concept behind this solution is to implement privacy between namespaces by introducing measures to enhance security. These measures must be extended to the parents' lists to ensure their effectiveness. If a measure is not extended in a parent when the namespace is closed, there will be no record of the code executed in that namespace. Each child namespace possesses a nPCR that is extended instead of using the standard PCR. To ensure privacy in this scenario, the measure extension in the parent will now focus on the current value of the nPCR rather than considering all the entries. To establish a Root of Trust for all these nPCRs, when an nPCR extension occurs, both the value and an IMA namespace identifier are provided. The namespace generates the event and then the options are two:

- The value of the template hash, along with the IMA namespace identifier, is sent directly to the host. The host then incorporates this value into its list and extends the true PCR accordingly. This simplifies the verification process, although it flattens the concept of the namespaces tree, losing the notion of father extension. However, the advantage lies in the ease of handling and managing the system.
- Every time an intermediate parent extends a measure, it sends the updated nPCR to its parent, eventually reaching the host. As a result, the nPCR is continuously extended up the parent chain. This approach ensures that when the integrity of a namespace needs verification, the entire parent chain is also verified along with it. This method provides a comprehensive verification process, encompassing all the relevant parent namespaces.

7.4.2 Implementation

Template

A new template was introduced to handle the insertion of nPCR values in the host's measurements list. Have two additional fields, one is the value of the nPCR extended and the other one is the IMA namespace identifier of the namespace that started the measure.

```
static const struct ima_template_field supported_fields[] = {
    ...
    {.field_id = "digev",
     .field_init = digest_namespace_event_init,
     .field_show = ima_show_template_digest},
    {.field_id = "imansid",
     .field_init = ima_id_init,
     .field_show = ima_show_template_string},
};

static struct ima_template_desc builtin_templates[] = {
    ...
    {.name = "ima-dig-imaid", .fmt = "digev|imansid"},
```

```
};
```

Listing 7.13. New IMA templates to save the nPCR values

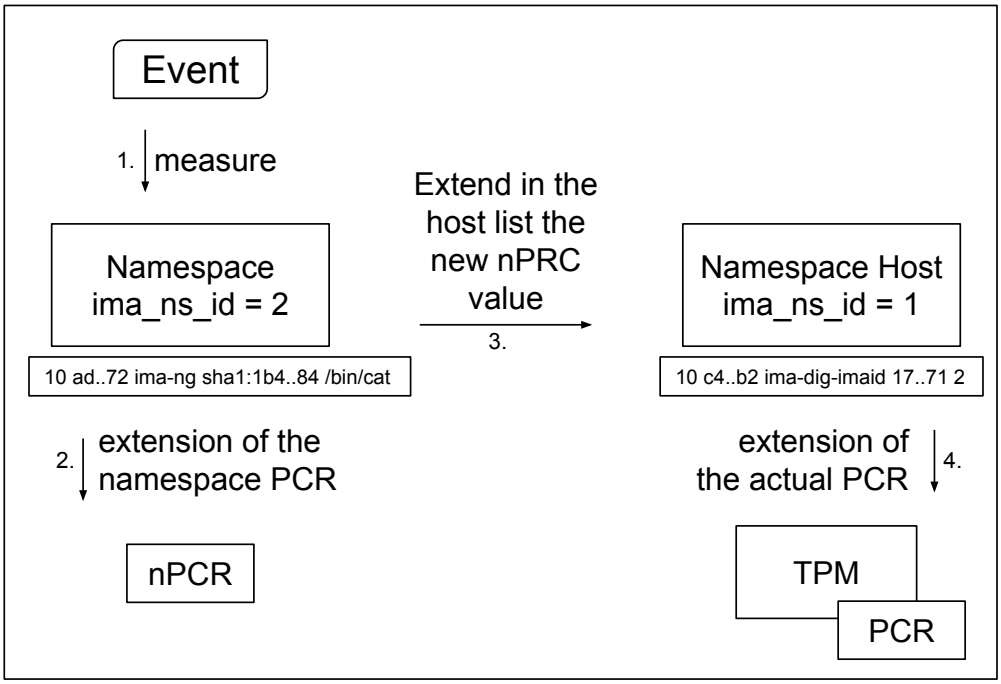

Figure 7.5. Schema for extension of nPCR

Extension of the nPCR

Whenever a measurement is taken in a child namespace, the PCR remains unchanged. Each namespace, beside the host, has its own namespace PCR (nPCR), which gets extended instead of the actual PCR. The nPCR is initialized with all zeroes. The extension process involves taking the current value of the nPCR, concatenating it with the measurement to be extended, and then hashing the result using the same hash function used for IMA. The resulting hash value becomes the new value of the nPCR. All is done acquiring the lock related to the nPCR. Below is the code that illustrates the nPCR extension process:

```
int vprc_extension(u8 *nPCR_value, u8 *value_to_extend) {
    int rc;
    SHASH_DESC_ON_STACK(shash, ima_algo_array[ima_hash_algo_idx].tfm);
    shash->tfm = ima_algo_array[ima_hash_algo_idx].tfm;
    rc = crypto_shash_init(shash);
    rc = crypto_shash_update(shash, nPCR_value, nPCR_MAX_LEN);
    rc = crypto_shash_update(shash, value_to_extend, nPCR_MAX_LEN);
    rc = crypto_shash_final(shash, nPCR_value);
    return rc;
}
```

Listing 7.14. Function to extend the nPCR value

Host extension

The nPCR must establish a chain of trust with the PCR that serves as the ultimate root of trust. After extending the nPCR, it is essential to safeguard it within the actual PCR. Subsequently,

the updated nPCR value, along with the IMA namespace identifier, is transmitted to the host. Upon receipt, the PCR is extended with this template hash. To achieve parent extension within this solution, the value is extended in the nPCR of the parent and then passed on to its parent namespace. The code below depicts the initialization process for the template entry and its subsequent insertion in the host's namespace.

```
void host_extension_nPCR(int start_ima_ns_id, u8 *template_digest) {
    ...
    event_data->ima_ns_id = start_ima_ns_id;
    event_data->template_start_digest = template_digest;
    ...
    template_desc->num_fields = template_num_fields;
    template_desc->fields = fields_const;
    result = ima_alloc_init_template(event_data, &entry, template_desc);
    result = ima_store_template(&init_ima_ns, entry,
               violation, NULL, NULL, PCR);
    ...
    return;
}
```

Listing 7.15. Function that saves the new value of the nPCR in the host's list

Mapping PID to IMA namespace identifier

For verification purposes may be useful to have the IMA namespace identifier available at the client side. This can be done using the proc filesystem [34]. Proc is a file-system mount point designed to offer access to the image of every running process in the system. Each entry in the `/proc` directory is represented by a five-digit decimal number that corresponds to the process ID. The ownership of each `file` is determined by the process's user ID, granting access permissions only to the owner (and only if the process's text file is readable). The size of a file corresponds to the total virtual memory size of the respective process.

A new entry has been added for the IMA namespace identifier. In the current implementation, the IMA namespace is created within the user namespace. For creating the necessary struct, all the fields are the same as those of the user namespace, except for the ID, which is the identifier specific to the IMA namespace. Since the IMA namespace is not stand-alone and does not have a `nscommon->inum`, for this field, the IMA namespace identifier is used.

```
static struct ns_common *imans_get(struct task_struct *task) {
    ...
    #ifdef CONFIG_IMA_NS
        if(user_ns->ima_ns != NULL)
            ima_id = ima_ns_id_from_user_ns(user_ns);
    #endif
    ima_ns_common->count = user_ns->ns.count;
    ima_ns_common->ops = user_ns->ns.ops;
    ima_ns_common->stashed = user_ns->ns.stashed;
    ima_ns_common->inum = ima_id;
    return ima_ns_common;
}
```

Listing 7.16. Additions to the procfs to handle the IMA namespace

By executing the command `ls -la` on the folder `/proc/{pid}/ns`, it is feasible to retrieve all the information concerning this process, including the details related to the IMA namespace identifier. The output will appear similar to the following format:

```
...
lrwxrwxrwx 1 root root 0 Aug 2 18:27 user -> 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 Aug 2 18:27 ima_ns -> 'ima_ns:[2]'
```

```
lrwxrwxrwx 1 root root 0 Aug 2 18:27 uts -> 'uts:[4026531838]'
```

Listing 7.17. New entry of the proc filesystem containing the ima_ns

Measurement list

In the measurement list, the entries coming from another namespace are inserted with the `ima-dig-imaid` template that includes the digest of the new value of the nPCR for the indicated namespace. Normal entries are the same as the default ima template `ima-ng`, with no needed additional fields.

```
10 1a...54 ima-ng sha1:1d...55 /usr/bin/cat
10 4c...1a ima-ng sha1:d5...23 /usr/bin/echo
10 b1...48 ima-dig-imaid d3...11 2
10 c4...61 ima-dig-imaid b8...16 3
```

Listing 7.18. Entries of the host's measurements list includes the nPCRs values of the other namespaces

7.4.3 Verification

This solution relies on excluding the measurements taken in the child namespaces from the host's list. Consequently, these values must be obtained from the user-space. The initial step involves validating the host's list by simulating the PCR extensions. When an entry originates from another namespace, it is treated as a distinct entry, and the simulation entails utilizing the template hash for extension. The final entry related to the namespace under verification will correspond to the latest nPCR value. After that depending on the solution:

- If the extension is performed directly in the host, the verification becomes simpler, as no additional lists are necessary. You only need to retrieve the list of the namespace under examination. Then, conduct the simulation of PCR extension and verify if the resulting value corresponds to the last nPCR value in the host list.
- If the parent extension has been executed, it is necessary to gather all the parents namespaces' lists. Starting from the list of the namespace itself, each subsequent list must be verified. The expected value of the nPCR for the namespace under examination is calculated by simulating the nPCR extension with its respective list. The parent list is then verified, and it is ensured that the last entry about the namespace under examination matches the value calculated from simulating the nPCR extension. This procedure is repeated iteratively until the host's namespace is reached.

To retrieve a namespace's list, you need to open the namespace's `ascii_measurements_list` from within the namespace. This can be accomplished using the proc filesystem [34]. In the proc filesystem, the list can be accessed at the path `/proc/{pid}/root/mnt/ima/ascii_runtime_measurements`. Here, it is possible to access every list associated with another namespace.

7.4.4 Strong points and weaknesses

The most important introduction to this solution is the concept of privacy between namespaces. This makes the host lose some control of namespaces, when a namespace is closed his list is deallocated, so is lost and is lost also the possibility to perform attestation on this namespace. Only the nPCR history is available but it has no meaning to know the code that runs on the namespace. To overcome the problem of losing the list of closed namespace a mechanism to save lists can be implemented. Is implemented, at the user level, a function to save the namespace's list to be accessible at verification even after the closure.

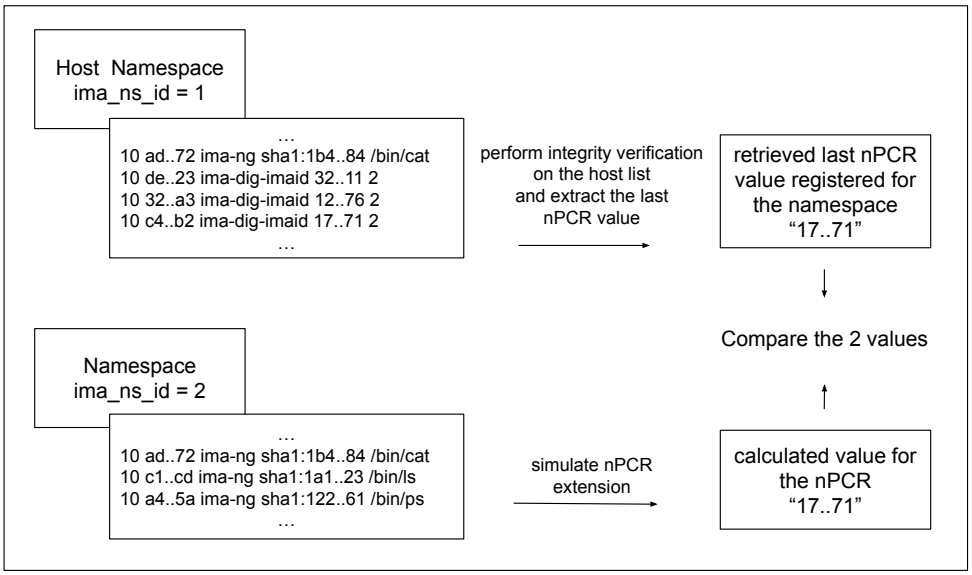

Figure 7.6. Verification of the nPCR solution

This solution allows for the attestation of a single namespace without accessing information about other namespaces. The attestation process relies solely on the historical values of the nPCR for each namespace. However, it's essential to note that when a container is attested, not all of the system is attested, which means the overall integrity of the entire host cannot be guaranteed. Verification is only performed on the programs directly running on the host, and entries from other namespaces remain unverified. The primary objective of this solution is to ensure the container's integrity, along with its dependencies, thereby preventing interference from programs running in other containers.

Chapter 8

Verification

The main aim of this solution is to provide container attestation. To verify the implementations, certain modifications were made to the Keylime framework [32]. Keylime is a TPM-based, highly scalable remote boot attestation, and runtime integrity measurement solution. It allows cloud users to monitor remote nodes using a hardware-based cryptographic root of trust. The specific modifications were implemented to accommodate new templates and logic for list verification.

8.1 Container virtualization

A container represents a distinct form of virtualization when compared to a Virtual Computer or Virtual Machine [35]. In those scenarios, all hardware components, encompassing processors, memory, and peripheral devices, are replicated by a software stratum referred to as a hypervisor or Virtual Machine Monitor. Within a Virtual Machine setup, access to physical hardware, such as network communication, hinges on interfaces presented by the hypervisor. Conversely, containers introduce a novel virtualization concept that doesn't encompass the entirety of a machine. This concept was conceived to enable the virtualization of applications that don't require the full spectrum of functionalities furnished by a Virtual Machine.

Google introduced Linux namespace kernel capabilities that paved the way for the emergence of this unique virtualization paradigm. Containerization elevates the level of abstraction, shifting from the hardware plane to the realm of the operating system. Containers eschew the need for their own virtualized hardware and instead directly engage with the underlying host kernel, leveraging the host system's hardware resources. This approach results in a significantly smaller footprint, measuring merely tens of megabytes, in stark contrast to the gigabytes occupied by a Virtual Machine. As a consequence, containerization demands less computational resources, leading to heightened operational speed due to the absence of hypervisor overhead.

8.1.1 Docker

Docker [37], an open-source framework, facilitates the development, deployment, and execution of applications within isolated process containers. The entirety of the requirements for an application to operate within a Docker container are encapsulated within an image. This image functions as a cohesive bundle of files, encompassing the complete "source code" of the Docker container.

Utilizing a client-server framework, the Docker platform is structured with the server-side being represented by the Docker daemon (`dockerd`). This daemon serves as the container runtime and is responsible for overseeing various Docker entities like images, containers, networks, and volumes. On the other hand, the client-side offers two primary options: the Docker Command Line Interface (CLI), which furnishes commands to execute operations on Docker elements, and Docker Compose, a specialized client designed for efficiently managing applications comprised of multiple interconnected containers.

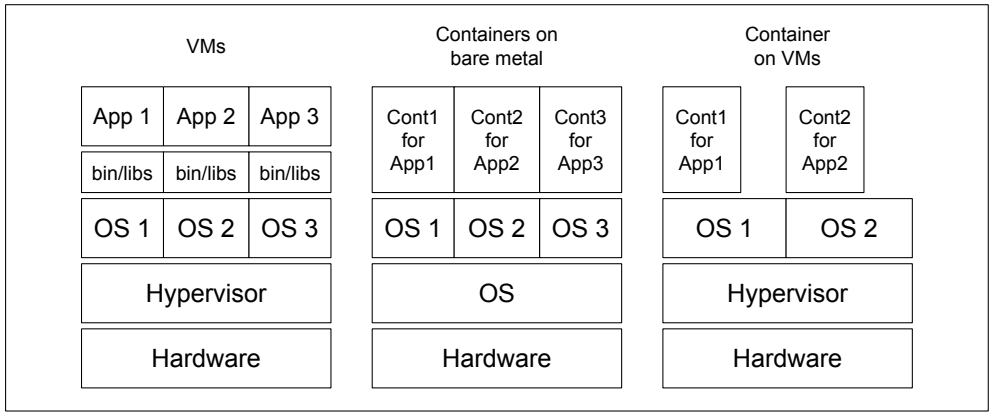

Figure 8.1. schema for container virtualization [36]

Docker Image

An image is a read-only template with instructions for generating a Docker container. Frequently, an image is derived from another preexisting image, incorporating further modifications. You have the option to produce your own images or exclusively utilize those crafted by others and made accessible in a registry. When crafting your custom image, you craft a Dockerfile, utilizing a straightforward syntax to delineate the sequence of actions necessary for image creation and execution.

Container

A container is a runnable instance of an image. Using the Docker API or CLI, you possess the ability to initiate, launch, halt, relocate, or erase a container. It's possible to affix a container to one or multiple networks, allocate storage to it, or even fabricate a novel image grounded in its existing condition. By default, a container showcases a certain level of isolation from both other containers and the host machine it resides on. You retain the capability to regulate the extent of isolation pertaining to a container's network, storage, and other foundational subsystems, whether in relation to other containers or the host machine. The essence of a container is shaped by its associated image, alongside any configuration parameters furnished during its inception or activation. Upon a container's removal, any alterations to its state that haven't been conserved in persistent storage vanish.

The execution of a container employs the Docker run command, which takes the image name as input to instigate container creation. If the image is absent locally, it is automatically fetched. Subsequently, the container is instantiated. Docker allows a read-write file system to serve as the container's ultimate layer, empowering the running container to forge or modify files and directories within its dedicated file system. Docker also establishes a network interface, linking the container to the default network and provisioning an IP address. By default, containers have the capacity to connect to external networks through the host machine's network connection. Additionally, specific flags or commands can be specified for execution during startup. For instance, flags such as `-t` facilitate terminal attachment, while `-i` enables interactive container operation.

8.2 Agent

Modifications were made based on the implementation of `keylime-rust` as described in Section A.3. However, the installation method remains unchanged.

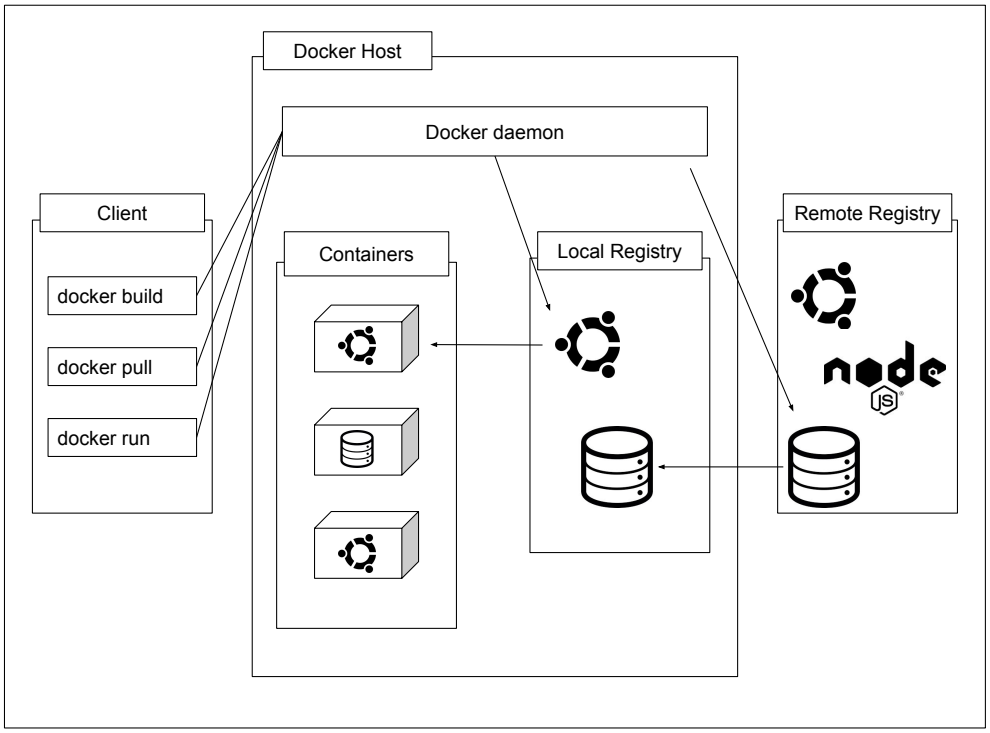

Figure 8.2. schema of the Docker architecture [37]

8.2.1 New APIs

The addition of a new API responsible for handling container attestation requests results in the retrieval of the following values:

- **Host's measurement list**: the whole host list, may be necessary to verify container dependencies and the nPCR value. In the other solutions is needed to verify the whole system.
- **IMA namespace identifier**: that corresponds to the IMA namespace created in the user namespace on which the container runs.
- **List of the namespace indicated**: used to verify the value of the nPCR in the third solution, otherwise is just a resume of what has been executed in the namespace.
- **Pubkey**: the public key related to the private key used to generate the quote.
- **Entry of the measurement list**: optional, used to trace the last entry of the list that has been verified.
- **TPM quote**: All the values generated by the TPM.
- **Algorithms**: information about the algorithms used to perform hash, encryption, and signing operations.

Requires the following parameters passed as request params:

- **nonce**: useful to avoid replay attacks [38]. A replay attack refers to a security breach where an attacker intercepts and replays messages from one context to another, tricking the legitimate participant(s) into believing they have successfully completed the intended security protocol or transaction. This manipulation of messages can deceive the system into accepting repeated or unauthorized actions, leading to potential vulnerabilities and unauthorized access;

- **mask**: to indicate the required PCRs for the attestation;
- **partial**: can be 1 or 0 in case of partial or not request;
- **ima measurement list entry**: last measured entry of this agent;
- **container id**: unique identifier of the container under verification.

8.2.2 Implementation

The function closely resembles the existing API `/quote/integrity`, which remains operational for standard IMA verification. In the first part of the function, it performs parameter checks and retrieves the measurement list. Additionally, there is a function call to map the container identifier to the corresponding IMA namespace identifier. Subsequently, various checks are executed, and finally, the quote is prepared and sent back as a response.

```
pub async fn integrity_container( req: HttpRequest,
    param: web::Query<Integ>, data: web::Data<QuoteData>,
) -> impl Responder {
    ...
    let (ima_id, ima_ns_list) = ima_ns_id_mapping(container_identifier);
    ...
    }
    let quote = KeylimeQuote {
        pubkey,
        ima_measurement_list,
        mb_measurement_list,
        ima_measurement_list_entry,
        ima_namespace_id,
        ima_measurement_list_namespace,
        ..id_quote
    };
    let response = JsonWrapper::success(quote);
    info!("GET integrity quote returning 200 response");
    HttpResponse::Ok().json(response)
}
```

Listing 8.1. Function to get the container data

8.2.3 Container identifier to IMA namespace identifier mapping

An essential step involves mapping the Container Identifier to the IMA namespace identifier. When a container is created with the user namespace mode enabled (as described in [39]), a user space is also generated, along with the corresponding IMA namespace. Identifying the namespace from the container identifier is crucial. This identification is facilitated by exposing the IMA namespace identifier in the user space, as detailed in section 7.4.2.

The script below outlines the steps for mapping and retrieving the namespace's measurement list using the proc filesystem. Initially, Docker commands [40] are employed to verify the container's existence on the host and obtain the PID of the container's main process. With the acquired PID, the script can access the proc filesystem to extract information regarding the IMA namespace identifier and the measurement list of the namespace.

```
...
dockerps=$(sudo -u lo Docker ps -a | grep -i "${firstchars}")
dockerval=$(sudo -u lo Docker container inspect $1 | grep -m 1 '"Pid": ')
docker_pid=$(echo "$dockerval" | awk -F':' '{print $2}')
pid=$(echo "$docker_pid" | tr -d ' ,')
output=$(ls -la /proc/$pid/ns | grep -i "imans")
```

```
ima_list=$(cat /proc/$pid/root/sys/kernel
       /security/integrity/ima/ascii_runtime_measurements)
tokens=( $output )
imans="${tokens[10]}"
OLD_IFS=$IFS
IFS=":" read -ra value <<< "$imans"
IFS=$OLD_IFS
nsbracket=${value[1]}
...
```

Listing 8.2. Script for container identifier to IMA namespace identifier mapping

8.3 Verifier

Creation of a new terminal command to handle the verifications of containers. The new command is `keylime_container_verification`.

8.3.1 Template

The keylime verifier requires a class for verification, specifically for each IMA template. During the simulation of the PCR extension, each entry is handled independently. A function is utilized to verify the template hash and extend the PCR with it. Additionally, there is a requirement for a function that performs checks on each component of the list entry. Keylime has introduced classes to manage all the new IMA templates that have been added for various solutions. Below is an example of the class for the nPCR solution, with the others being similar but dealing with different fields.

```
class ImaDigNs(Mode):
    digest: str
    ima_ns_id: int
    def __init__(self, data: str):
        tokens = data.split(" ", maxsplit=1)
        ...
        self.digest = str(tokens[0])
        self.ima_ns_id = int(tokens[1])

    def bytes(self) -> bytes:
        return str.encode(self.bytes) + str.encode(str(self.ima_ns_id))

    def is_data_valid(self, validator: Validator) -> Failure:
        return validator.get_validator(type(self))(self.digest, self.path,
            self.ns_id, self.counter)
```

Listing 8.3. Class for handling the new template

8.3.2 Counter solution Verification

This initial solution is the simplest to validate. When simulating the PCR extension, if an entry from another namespace is encountered, it becomes essential to regenerate all entries from the intermediate namespaces. The counter, which tracks the number of performed PCR extensions, enables the reconstruction of all other entries, commencing from just the entry in the host namespace. For example the entry:

```
10 16..5f ima-nsid-cnt sha1:8d..19 /bin/cat 7 4
```

This indicates that the namespace with the IMA namespace identifier "7" has generated an event that extended the PCR four times. Therefore, all the entries related to this event across all namespaces, starting from the namespace that triggered the event and going up to the host, are as follows:

```
10 16..5f ima-nsid-cnt sha1:8d..19 /bin/cat 7 1
10 9d..f6 ima-nsid-cnt sha1:8d..19 /bin/cat 7 2
10 1c..12 ima-nsid-cnt sha1:8d..19 /bin/cat 7 3
10 58..a8 ima-nsid-cnt sha1:8d..19 /bin/cat 7 4
```

This logic has been implemented in Keylime in this way. It's important to highlight that it is necessary to recalculate the appropriate template hash for each new entry generated, ensuring the accurate value for simulating the PCR extension.

```
if int(tokens[-1]) > 1 and tokens[2] == "ima-nsid-cnt":
    number_of_extensions = int(tokens[-1])
    for x in range(1,number_of_extensions):
        value_4 = tokens[-4].split(':')
        byte_value = codecs.decode(value_4[1], 'hex')
        value_sha1 = value_4[0] + ':' +"\x00"
        value_to_hash = bytearray(value_sha1.encode('utf-8')) + byte_value
        value_3 = tokens[-3] + '\x00'
        value_to_hash = value_to_hash + bytearray(value_3.encode('utf-8'))
        value_to_hash = value_to_hash + bytearray((tokens[-2] +
            str(x)).encode('utf-8'))
        value_hashed = Hash.compute_hash(hash_alg, value_to_hash)
        runninghash = Hash.compute_hash(hash_alg, runninghash + value_hashed)
```

Listing 8.4. PCR extension simulation for the entries with the new template

8.3.3 Event solution Verification

The primary concept behind the verification process is similar to the Counter solution mentioned in 8.3.2, which involves determining how many times the PCR has been extended for a specific event. However, in this case, there is no counter involved. Taking advantage of the record of namespace creation, it becomes feasible to construct a tree representing the sequence of namespace creations:

```
if tokens[2] == "ns-event":
    if str.isdigit(tokens[4]) and str.isdigit(tokens[5]):
        parent = int(tokens[4])
        child = int(tokens[5])
    else:
        break
    if tokens[3] == 0:
        tree.add_child_between(tree.find_in_childs(parent),
            tree.find_in_childs(child))
    elif tokens[3] == 1:
        tree.close_ns(child)
    else:
        break
```

Listing 8.5. Handle of the entries regarding the creation of closure of a namespace

An ad-hoc library has been crafted to manage the creation of trees and perform operations on them. This library stores the namespace identifier as supplementary data within each tree node. Determining a namespace's position within the tree enables the calculation of the node's height. Consequently, it becomes feasible to know the number of intermediate nodes between the namespace under examination and the host. Thus, the height of the node functions as a counter for tracking the number of PCR extensions carried out for a particular event.

```
if tokens[2] == "ima-id":
    number_of_extensions = tree.node_heigh(tree.find_in_childs(tokens[5]))
    hash_mes = tokens[3].split(':')
    byte_value = codecs.decode(hash_mes[1], 'hex')
    value_sha1 = hash_mes[0] + ':' +"\x00"
    value_to_hash = bytearray(value_sha1.encode('utf-8')) + byte_value
    file_path = tokens[4] + '\x00'
    value_to_hash = value_to_hash + bytearray(file_path.encode('utf-8'))
    value_to_hash = value_to_hash + bytearray((tokens[-1]))
    value_hashed = Hash.compute_hash(hash_alg, value_to_hash)
    for x in range(0,number_of_extensions):
        runninghash = Hash.compute_hash(hash_alg, runninghash + value_hashed)
```

Listing 8.6. Handle the entries regarding a measure

8.3.4 Find the nPCR value

To validate the list of namespaces received from the host, we must extract the final nPCR value from the host's list. The nPCR value is safeguarded in the host's list using the true PCR extension. To obtain this value, we need to scroll through the host's list and retrieve the last entry associated with the IMA namespace identifier that is currently under verification. The function shown given the whole measurement list and an IMA namespace identifier will return the last nPCR value related to the chosen namespace.

```
def find_last_nPCR(data: str, ns_id: int) -> str:
    last_val: str = "";
    data_arr = data.split("\n")
    for element in data_arr:
        if element != "":
            tokens = element.split(" ")
            if int(tokens[4]) == ns_id:
                last_val = tokens[3]
    return last_val
```

Listing 8.7. Retrive the last nPCR value regarding to a given namespace

8.3.5 Verfication with nPCR

Once we have obtained the last nPCR value of the namespace under verification, it is necessary to perform the verification process. Since the list format is identical to the host's format, we can call the appropriate function to verify the list using the namespace's list and the nPCR quote as parameters. However, this verification is only performed if the response contains the IMA namespace identifier; otherwise, it proceeds as a normal attestation. If any errors occur during the list verification process, the failure is logged or recorded accordingly. When a container is verified all the namespaces created are also verified. Docker gives the possibility to retrieve all the information regarding the process created by the container and if all the lists are available then are verified.

```
if "ima_namespace_id" in agent.keys() and "ima_measurement_list_namespace" in
    agent.keys():
    quote_vpcr = find_last_nPCR(agent["ima_measurement_list"],
        agent["ima_namespace_id"])
    ima_measurement_list_namespace = agent["ima_measurement_list_namespace"]
    quote_validation_failure = get_tpm_instance().check_quote(
        agentAttestState, agent["nonce"], received_public_key, quote_vpcr,
        agent["ak_tpm"], agent["tpm_policy"], ima_measurement_list_namespace,
        runtime_policy, algorithms.Hash(hash_alg), ima_keyrings,
        mb_measurement_list, agent["mb_refstate"],
        compressed=(agent["supported_version"] == "1.0"))
```

```
failure.merge(quote_validation_failure)
```

Listing 8.8. Verification of the list with nPCR

8.3.6 Save namespace lists

In the existing implementation of the IMA namespace, when a namespace is closed, its list is lost, making it challenging to perform attestation after the closure. To enable the attestation of a namespace even after it is closed, it is essential to preserve the list. Addressing this issue requires exploring various solutions, each operating at different levels of the system.

Kernel side

The concept involves preserving the list by triggering the preservation process when the function responsible for destroying the IMA (Integrity Measurement Architecture) namespace, named `ima_free_ima_ns`, is invoked. However, this approach faces a challenge as the file needs to be stored outside of kernel memory. To address this issue, the current direction in the Linux community is to develop various filesystems that facilitate communication between the kernel and user space. One such filesystem is "securityfs," which enables access to information in kernel memory.

Since kernel information is dynamic and can change, the approach taken is to dump the information into files when these files are read. Each file in these special filesystems requires a set of operations, represented by a custom struct of functions that handle read, write, and other operations. When a file is opened, the corresponding read function is invoked, and the file is populated with the relevant information. For example, this struct is the one used for the creation of the file `ascii_runtime_measurement`.

```
static const struct file_operations ima_ascii_measurements_ops = {
    .open = ima_ascii_measurements_open,
    .read = seq_read,
    .llseek = seq_lseek,
    .release = seq_release,
};
```

Listing 8.9. Struct to handle securityfs operations

Within securityfs, integration with IMA templates takes place, where each template field possesses a specific function responsible for writing its data to a file in securityfs. However, a challenge arises because these files are intended for accessing kernel information and are already readily available, making it difficult to copy a file at the kernel level. To overcome this obstacle, customized functions must be provided to manage the diverse templates. Although this approach is not quick to implement, the Linux community aims to limit file access from the kernel level [41].

Currently, certain services, such as logging in the Linux kernel, no longer utilize regular files; instead, they rely on the `dmesg` command [42]. This command enables the dumping of the `syslog`, and to prevent kernel memory saturation, it automatically starts deleting the earliest entries when the log reaches a certain size. This ensures the log remains manageable and doesn't consume excessive memory resources.

Certain user-level applications have been developed to copy the contents of the `syslog` using the `dmesg` command and store it in history files at the user level. The primary purpose of these applications is to maintain a comprehensive record of the kernel log history, ensuring that crucial information and events are preserved for analysis or troubleshooting purposes. By copying and saving the `dmesg` output in user-level files, users can access and review past kernel log entries at their convenience without directly accessing the kernel memory or interfering with the kernel's logging process. This approach offers a practical and non-intrusive way to retain a complete historical record of the kernel's activities.

Periodical checks

A solution has been developed to adapt the Linux kernel to its current vision by collecting data on the user-side. To address this, a modified approach is proposed to store syslog data in user memory, particularly for the measurement list. This involves the creation of a script that periodically copies the `ascii_runtime_measurements` file into user memory. However, a challenge arises due to the `syslog` being a circular buffer, leading to the loss of older data when new data arrives.

For successfully saving the `syslog`, the system needs to keep track of the last recorded entry and append new data to the file. However, the issue becomes more complex when dealing with saving a namespace list. When a namespace is closed, the list gets deleted, and there is a possibility of losing some entries. If an entry is recorded, and the namespace is closed before the script saves it in the user file, that entry is lost.

Finding the right compromise for the data collection interval is challenging. If too much time passes between two data collections, there is a higher probability of losing entries. On the other hand, collecting entries too frequently negatively impacts performance since the list needs to be repeatedly dumped from the kernel side and copied into user memory.

When certain elements in a list are lost, the ability to effectively verify the namespace is compromised. This emphasizes the impracticality of the solution, as even a low probability of losing an entry renders the solution unusable. However, if this issue only affects already closed containers, the solution might still be considered as an option when no better alternatives are available.

Container script

To address the challenge of conducting periodic checks, the approach involves transferring the responsibility of saving the list to a script that runs during container startup. With this setup, each container runs a script capable of capturing signals indicating the imminent closure of the container, enabling timely saving of the list. The script effectively captures the signals `SIGTERM` and `SIGINT` [43].

The `SIGTERM` signal serves as a request to terminate a process. Unlike the `SIGKILL` signal, which forces immediate termination without recourse, `SIGTERM` can be caught and interpreted or disregarded by the process. This affords the process the opportunity to gracefully terminate, freeing up resources and preserving its state, if necessary. The `SIGINT` signal closely resembles `SIGTERM` and is sent by the controlling terminal when a user desires to interrupt the program, often accomplished by pressing `Ctrl + C`, for instance.

The trap command [44] handles the receive of indicated signals. In this case, `SIGTERM` and `SIGINT` are captured. This script needs to be run in the container in the background. Is able to capture the closure of the container and copy the list before.

```
exit_script() {
    # perform the copy of the list in user space
    exit 0
}
trap exit_script SIGTERM SIGINT
while true; do
    sleep 1
done
```

Listing 8.10. Script that handles the save of the container's list upon its closure

In order to run the script in the container this can be done using a Dockerfile [45] to create the container image. One of the ways to make a script execute at container start is to use `ENTRYPOINT`, so the Dockerfile will contain the following entries. The Dockerfile can also contain commands to activate the IMA namespace. However, it's important to note that in the current implementation, the IMA namespace is not allocated by default within the user namespace for compatibility reasons, as explained in section 5.2.

```
FROM ubuntu:latest
COPY script_ubu.sh /
RUN chmod +x /script_ubu.sh
RUN mount -t securityfs /mnt /mnt
RUN echo 1 > /mnt/ima/active
ENTRYPOINT ["/bin/bash", "/script_ubu.sh", "&"]
```

Listing 8.11. Dockerfile example for run the script when the container starts

To create a Docker image from a Dockerfile and a specified "context," you must use the Docker build command as follows [46]. The build process involves referencing files in the specified PATH or URL, allowing the Dockerfile to utilize any of these files. For instance, you can employ a COPY instruction to reference a file within the context. To build a Dockerfile and run a container, execute the following commands:

```
$ docker build --tag image_name .
$ docker run -i -t --cap-add sys_admin image_name
```

This solution streamlines the collection of the namespace's lists while mitigating the risk of losing any entries. However, delegating the responsibility of list collection to a script that must be executed within the container grants excessive power to the user. The script has to capture all the lists, also coming from namespaces created by the container. Docker provides comprehensive information about the processes it creates. However, a challenge arises when a user namespace is established and linked to an existing process. In such scenarios, no new process is generated, leading to a situation where the trap responsible for saving the list upon process closure won't be triggered, resulting in the loss of the list.

A more robust approach is to integrate the list collection directly into the Docker daemon. Although this implementation is more challenging and requires compatibility with the standard implementation, it offers a better solution. As the IMA namespace is not currently a standalone namespace, any new features must be developed in accordance with the evolution of IMA namespaces. This ensures a more reliable and secure method of list collection and management.

Chapter 9

Tests

The section details the outcomes of the examinations conducted on the Keylime framework, which underwent adjustments to facilitate container attestation and kernel modifications. Specifically, operational tests were enacted to validate the accurate functioning of the attestation procedure involving the Trust Monitor and Keylime frameworks and the measure operation. Additionally, performance assessments were carried out to appraise the time delays and resource utilization during the attestation process and the registration of an event.

9.1 Testbed

The assessment of the operational efficiency and effectiveness of the suggested solutions involves executing tests on a system equipped with an Intel i5-5300 processor clocked at 2.30 GHz, featuring 4 cores, 8 threads, 16 GB of RAM, and a TPM2.0 chip. This system runs on Ubuntu 22.04 LTS and operates with a custom Linux kernel derived from version 6.0. The environment includes Docker version 24.0.5, configured with namespacing, alongside a customized version of Keylime 7.0. This version of Keylime has been adapted to facilitate container verification.

The attester and verifier share the same host machine. For instructions on kernel compilation and installation for the testbed configuration, please refer to Appendix A. You can find steps for installing and configuring Keylime in Appendix A.5.6 and instructions for installing Docker and configuring it in Appendix A.6.

9.2 Functional tests

Functional tests aim to confirm whether the software's implementation aligns with the specified solution's requirements.

9.2.1 Measure Test

This illustrates the process of saving measurements in different lists depending on the solution. All the tests involve the execution of identical programs within the containers. The containers execute `ls` and continuously `sleep` command in a loop until the container is closed. For the third solution, as described in Section 7.4, the list is saved upon closing. However, in the first two solutions, there is no need to save the container's list, as all the information is already present in the host's list.

Once the container has been executed, it is possible to access the container's list through the `/proc` filesystem. By using the command `$ docker inspect container_name`, is possible to obtain information about the primary process of the container. The list can be accessed at the following path:

```
/proc/{pid_container}/root/mnt/ima/ascii_runtime_measurements
```

Counter Solution 7.2

This solution needs to set the new IMA template `ima-nsid-cnt` by doing:

```
$ vim /etc/default/grub
# change the GRUB_CMDLINE_LINUX adding
# with ima_template=ima_nsid_cnt
$ sudo update-grub
# restart the system
# verify by checking  /proc/cmdline
```

Otherwise is possible to select a compile time in the `menuconfig` under `ima_template` entry

The container is running with a script that performs late-init of the IMA namespace containing only those lines

```
$ mount -t securityfs none /mnt
$ echo 1 > /mnt/ima/active
$ echo -e 'measure func=BPRM_CHECK\nmeasure func=FILE_MMAP mask=MAY_EXEC\n' \
  > /mnt/ima/policy
```

The container's list will include two entries. The `"2"` represents the IMA namespace identifier of the container, and the `"1"` signifies the counter value. Since this is the container that generated the event, the counter value is one because the PCR has been extended only once for this specific event at this point in time.

```
10 d9..ea ima-nsid-cnt sha1:c6..aa /bin/ls 2 1
10 c1..b4 ima-nsid-cnt sha1:dc..66 /bin/sleep 2 1
```

Listing 9.1. Entry of the namespace's list with the counter solution

The parent extension has been correctly performed so is possible to see the measure effectuated by the container in the host's list. Since the host registered the event in its list the PCR has been re-extended and so the counter has been incremented. The last entry reported regards a measure effectuated in the host so have `"1"` as IMA namespace identifier and `"1"` as the counter.

```
...
10 17..4c ima-nsid-cnt sha1:c6..aa /bin/ls 2 2
10 9a..73 ima-nsid-cnt sha1:dc..66 /bin/sleep 2 2
10 92..18 ima-nsid-cnt sha1:1e..e0 /bin/cat 1 1
...
```

Listing 9.2. Entry of the host's list with the counter solution

All the events were correctly inserted in the lists, the counter is incremented consistently. The events generated by the host are correctly handled.

Event Solution 7.3

Additionally, this solution necessitates configuring the 'ima_template' just as it was done for the Counter solution (see Section 9.2.1). However, this time, the template name should be `ima-id`. Furthermore, you will need the same script for late initialization as used in the Counter solution. The template utilized for registering the creation or closure of a namespace (`ns-event`) is automatically applied to manage these events and does not require explicit specification.

Every entry in the lists will include an additional field indicating the IMA namespace identifier. In this scenario, all entries in the list of the container's parents will be identical. The list of the containers generating an event will appear as follows:

```
10 c9..47 ima-id sha1:c6..aa /bin/ls 2
10 11..48 ima-id sha1:dc..66 /bin/sleep 2
```

Listing 9.3. Entry of the namespace's list with the event solution

The host's list will record information related to the creation of namespaces. In this context, the initial field of each entry is employed to signify whether it pertains to the creation `(0)` or closure `(1)` of a namespace. The remaining two fields denote the IMA namespace identifier of the parent of the targeted namespace and the namespace that is the focus of the event. It is possible to differentiate entries originating from other namespaces through the additional field incorporated into all measurement entries. Within this list, you can observe the log entries denoting the creation of the namespace with IMA namespace ID `2`, the events it generates, and its closure. Given that all the fields in these measure entries match those in the namespace's list, entries from other namespaces will also be documented here without any alterations. This feature proves highly advantageous during the verification process.

```
...
10 99..27 ns-event 0 1 2
10 c9..47 ima-id sha1:c6..aa /bin/ls 2
10 11..48 ima-id sha1:dc..66 /bin/sleep 2
10 4c..a5 ns-event 1 1 2
...
```

Listing 9.4. Entry of the host's list with the event solution

The events associated with namespace creation and closure are appropriately recorded, and all events within the container's namespace are extended including the container's identifier. This enables the reconstruction of the namespace creation tree as represented in Figure 7.4.

nPCR Solution 7.4

This solution eliminates the need to define a new IMA template; instead, it utilizes an existing template called `ima-dig-imaid` for registering the values of the nPCR in the parent list, and this template is automatically applied when necessary. The solution relies on the standard `ima-ng` for the measure events, for this reason, the list of a container will look like a standard IMA list.

It is imperative to incorporate logic into the script that runs within the container to facilitate list saving upon closure. Failing to do so would render verification impossible after the container has terminated. Details on the script are found in the Appendix A.6. Additionally, it is essential to specify the correct path for saving the list; it is recommended to save it in a directory protected at the root level.

```
10 1d..44 ima-ng sha1:c6..aa /bin/ls
10 b6..2c ima-ng sha1:dc..66 /bin/sleep
```

Listing 9.5. Entry of the namespace's list with the nPCR solution

Since the container's namespace will not extend the physic PCR but a virtual one to create a root of trust when the nPCR is extended the new value is saved in the parent's list. In this scenario, the host maintains a record of the nPCR values for the container's namespace. For the two measurements conducted within the container, there are two nPCR extensions in the host's list, each associated with the IMA namespace identifier of the container's namespace. The final entry reported is a host measurement, which employs the default IMA template.

```
...
10 c9..47 ima-dig-imaid 1D..44 2
10 11..48 ima-dig-imaid F8..2C 2
10 54..c7 ima-ng sha1:40..ff /bin/cat
...
```

Listing 9.6. Entry of the host's list with the nPCR solution

The host list compiles comprehensive data regarding the historical nPCR values within namespace identifier 2. However, it's important to note that this list solely provides nPCR values and does not include any information about the specific programs executed within the namespace. The entries related to measures in the host list exclusively pertain to events generated by the host itself.

9.2.2 List verification Test

These tests involve verifying the container list used for Test 9.2.1. We will examine the agent's API and the command for initiating container verification. The container execution is minimal, so the whitelist will only include the `sleep` and `ls` commands.

The Agent's API is accessible through `/quotes/container`, and for quote attestation of a container, in addition to the standard parameters (nonce, mask, sign algorithm, hash algorithm, public key, and other optional parameters), the container identifier must be provided as part of the request. The host housing the container has been successfully registered with the Keylime registrar. Additionally, the API will return the container list along with all the standard values related to PCR, ensuring message integrity through the TPM's signature.

For starting the verification of a given container is possible to call it using the terminal command, the container will be verified and its status stored.

```
$ sudo keylime_verify_container {container_id}
```

To independently test the API call, we utilize the Postman [47] program, which is an API platform designed for constructing and utilizing APIs. The API path is always the same, depending on the IMA template found in the list uses different verification logic. In this case, it is employed to make the API call. The request will be structured as follows:

```
"GET",
http://192.168.0.109:3310/v2/quotes/container
?nonce=5437436253412&partial=1&ima_ml_entry=0&containerid=432432
```

A brief explanation of the fields present in the response given by the Agent's API for the container integrity:

- `ima_measurement_list`: contains the whole host's list;
- `mb_measurement_list`: measured boot information, this field is retained to ensure compatibility with the standard quote integrity of the host's list; however, for container verification, it will consistently be set to `None`;
- `ima_measurement_list_entry`: index of the latest entry in the host's list that has been verified for integrity;
- `ima_namespace_id`: IMA namespace identifier of the namespace associated at the container under verification;
- `ima_measurement_list_namespace`: list of the namespace associated with the container under verification;
- `tpm_quote`: TPM values for PCRs, signed by the TPM with information about the algorithms used;
- `other info`: there is other information about the algorithms used, public keys, and other fields useful for verification.

Counter Solution 7.2

After the call of the API, the return value will be:

```
{
    "ima_measurement_list": "...10 17..4c ima-nsid-cnt sha1:c6..aa
    /bin/ls 2 2\n10 9a..73 ima-nsid-cnt sha1:dc..66 /bin/sleep 2
    2\n10 92..18 ima-nsid-cnt sha1:1e..e0 /bin/cat 1 1..."
    "mb_measurement_list": None,
    "ima_measurement_list_entry": 2,
    "ima_namespace_id": 2,
    "ima_measurement_list_namespace: "10 d9..ea ima-nsid-cnt
    sha1:c6..aa /bin/ls 2 1\n10 c1..b4 ima-nsid-cnt sha1:dc..66
    /bin/sleep 2 1",
    "tpm_quote": {
        "nonce": ...
        "pcrs": ...
        ...
    }
    "hash_algo": "sha256"
    "enc_algo": "aes-256-CBC"
    "sign_algo": ...
    "public_key" ...
    ...
}
```

Listing 9.7. Response of the container verification with the counter implementation

It encompasses all the essential information from the namespace list required for the verifier to conduct integrity verification. Once the verifier is executed with the appropriate container whitelist for conducting container verification, it successfully records the container's integrity status without any issues. The verifier proceeds to simulate the extension of the container list and confirms its alignment with the retrieved PCR value. All of this occurs subsequent to the verification of the TPM's signature on the quote. If the container list is incomplete or contains elements not found in the whitelist, an error message will be generated and displayed.

Event Solution 7.3

The verification process is always the same, since the changes are only in the verification logic, launching the same API the result will be:

```
{
    "ima_measurement_list": "...10 99..27 ns-event 0 1 2\n10
    c9..47 ima-id sha1:c6..aa /bin/ls 2\n10 11..48 ima-id
    sha1:dc..66 /bin/sleep 2\n10 4c..a5 ns-event 1 1 2..."
    "mb_measurement_list": None,
    "ima_measurement_list_entry": 2,
    "ima_namespace_id": 2,
    "ima_measurement_list_namespace: "10 c9..47 ima-id sha1:c6..aa
    /bin/ls 2\n10 11..48 ima-id sha1:dc..66 /bin/sleep 2",
    "tpm_quote": {
        "nonce": ...
        "pcrs": ...
        ...
    }
    "hash_algo": "sha256"
    "enc_algo": "aes-256-CBC"
    "sign_algo": ...
    "public_key" ...
```

```
    ...
}
```

Listing 9.8. Response of the container verification with the event implementation

In this approach, the verification process involves generating a namespace creation tree to reconstruct the precise number of physical PCR extensions for each event. When invoking the `keylime_verify_container` with the container whitelist, it will log the container's status.

nPCR Solution 7.4

In this situation, it's possible that the container has been terminated, necessitating the retrieval of the list from the location specified during the container's launch. The default location for this is `/root/namespaces`. The response from the API call is as follows:

```
{
    "ima_measurement_list": "...10 c9..47 ima-dig-imaid
    1D..44 2\n10 11..48 ima-dig-imaid F8..2C 2 \n10 54..c7
    ima-ng sha1:40..ff /bin/cat..."
    "mb_measurement_list": None,
    "ima_measurement_list_entry": 2,
    "ima_namespace_id": 2,
    "ima_measurement_list_namespace: "10 1d..44 ima-ng
    sha1:c6..aa /bin/ls\n10 b6..2c ima-ng sha1:dc..66
    /bin/sleep",
    "tpm_quote": {
        "nonce": ...
        "pcrs": ...
        ...
    }
    "hash_algo": "sha256"
    "enc_algo": "aes-256-CBC"
    "sign_algo": ...
    "public_key" ...
    ...
}
```

Listing 9.9. Response of the container verification with the nPCR implementation

To achieve this, the solution entails simulating the extension of the nPCR for the container and obtaining the latest nPCR value from the host's list following its verification. The functioning of `keylime_verify_container` remains consistent with previous solutions. In this case, as well, no error message is generated when providing a container whitelist that includes the `ls` and `sleep` commands. However, in other cases, an error message is generated.

9.3 Performance tests

In this section, the tests are designed to assess performance in terms of execution time and resource utilization. These tests involve running varying numbers of containers on the same machine, with each container carrying out more complex operations. This evaluation is particularly crucial for verifying the execution time of the first two solutions when measuring an event. In these two solutions, the parent extension is executed atomically to prevent another container from extending the PCR while a parent extension operation is in progress. This precaution can result in longer waiting times on the lock.

The expectation is that the nPCR solution will demonstrate faster performance compared to the other two solutions when measuring an event. This is primarily because the physical PCR is extended only once, and the parent extension is not atomic. Since the PCR is extended only

once, there is no need to lock everything to maintain the order of parent extensions. The critical aspect is ensuring that the new nPCR value is accurately stored in the parent's list.

However, in a containerized environment, namespace chains are typically quite short. In most instances, only the host creates namespaces for other containers, resulting in chains with a length of just two. With these shorter chains, differences in execution time may not be very noticeable. Additionally, the time spent on acquiring the lock may only become problematic when dealing with longer namespace chains. This is because acquiring the lock is necessary to perform the entire parent extension, potentially causing the lock to be held for longer periods.

During the verification process, it is expected that all solutions will have comparable execution times, as the most time-consuming operation in the verification process is the generation of the TPM quote, which is a common step across all solutions.

9.3.1 Measure Performance

Here is shown the time required to execute a measurement for each solution. The tests are conducted using varying numbers of containers to assess the lock's waiting time. All containers will utilize a modified Ubuntu image that generates a new file in a loop, each containing a fixed random value. Additionally, these containers will execute the `cat` command on the newly created files to facilitate measurement. This approach ensures a consistent, even if not profoundly impactful, time for hashing the files. Container policies have been configured to exclusively measure the newly created files. The reported execution times are all calculated as averages across various measurements, allowing us to determine the average execution time. This approach is particularly relevant for the first two solutions, as the waiting time for the lock can vary.

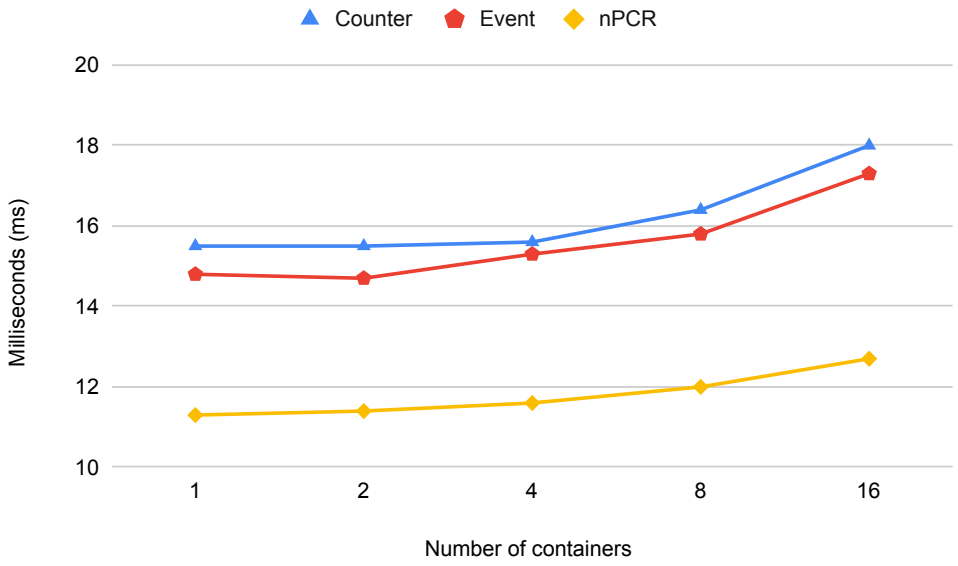

Figure 9.1. Comparison of the IMA measure latency in different solutions in milliseconds (ms)

As anticipated, the solution employing nPCRs demonstrates slightly superior speed, due to its utilization of a single physic PCR extension, in contrast to the two PCR extensions employed by the other two solutions. The time required increases as the number of containers grows, primarily due to the higher concurrency required for lock acquisition. These tests were conducted using a container designed to stress the measurement infrastructure by consistently generating new files and measuring them.

In a typical container environment, we anticipate less strain on the measurement infrastructure because containers typically conduct the bulk of their measurements shortly after booting, followed

by a reduction in measurement frequency. The tests were conducted using an Ubuntu image that continuously generates new files to exert significant stress on the system. This approach was employed to ensure that the time required to acquire the lock remains finite and to underscore the superiority of the solution that doesn't necessitate a complete lock on the parent extension, particularly in high-load scenarios.

In Figure 9.1, the graph demonstrates that as the number of containers increases, all solutions exhibit deteriorating performance. This degradation is primarily attributed to the insertion process into the host's list. When inserting a value into the host's list, a lock is employed, a practice also found in the standard IMA implementation, to guarantee the atomicity of the insertion into the list and the PCR extension. Consequently, in the nPCR solution, when it becomes necessary to insert a new nPCR value into the host list, acquiring a lock becomes imperative, a lock shared among all the namespaces engaged in this operation as well as the host itself. Fortunately, this lock entails relatively shorter waiting times, and as a result, the nPCR solution experiences a more modest increase in its performance degradation curve as the number of containers grows.

9.3.2 Verification Performance

The evaluation of the Verification process will encompass an assessment of CPU usage and RAM consumption on the attesting platform. This assessment will be conducted both with and without the remote attestation process. It also performed an analysis of the time required for the verification cycle. Is expected that there won't be significant variations in attesting times among the different solutions. The most time-consuming task in the verification process is generating the TPM quote, which is a standard operation across all solutions. Therefore, the differences between them are not expected to be substantial.

All measurements are averages taken from multiple attestation processes to ensure meaningful results on machines running for ten minutes. It is intuitive that the verification process consumes fewer resources and takes less time on systems that have been recently started, as they have a shorter measurement list.

CPU and RAM usage

Here are the graphs representing CPU (Figure 9.3) and RAM (Figure 9.2) utilization, obtained from the attesting platform. All data is presented in percentage values. The RAM and CPU usage slightly increases with the increase of the containers because a longer list is necessary to be verified.

Attestation time

The graph in Figure 9.4 illustrates that there is not a significant differentiation in values, as the most time-consuming process, the creation of the TPM quote, takes about one second and is consistent across all solutions.

Another important thing to consider when evaluating the attestation cycle latency is the machine up-time. The longer the host's list the longer will be the verification process because there are more entries in the list that needs to be verified. Keylime has an option to start the verification of a list from a given entry, the previous part of the list has been verified in the past. So performing periodic verification the attestation time will be constant but performing attestation on a machine after a long period will lead to a longer attestation time.

9.3.3 Comparisons with other solutions

The solution, for container attestation, currently utilized in the "TORSEC" laboratory has a different approach and does not rely on IMA namespaces. Instead, it leverages the functionality of `cgroups` [27] to identify containers at the kernel level. This method makes use of the `cgroup identifier` assigned during container creation. It exclusively depends on the host's list, ensuring

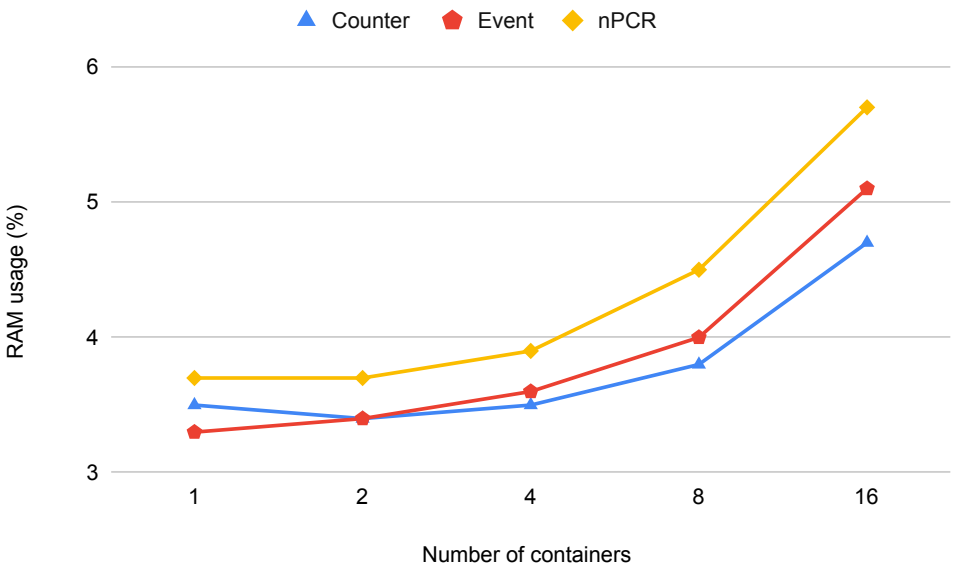

Figure 9.2. Percentages of the RAM usage in the attesting platform

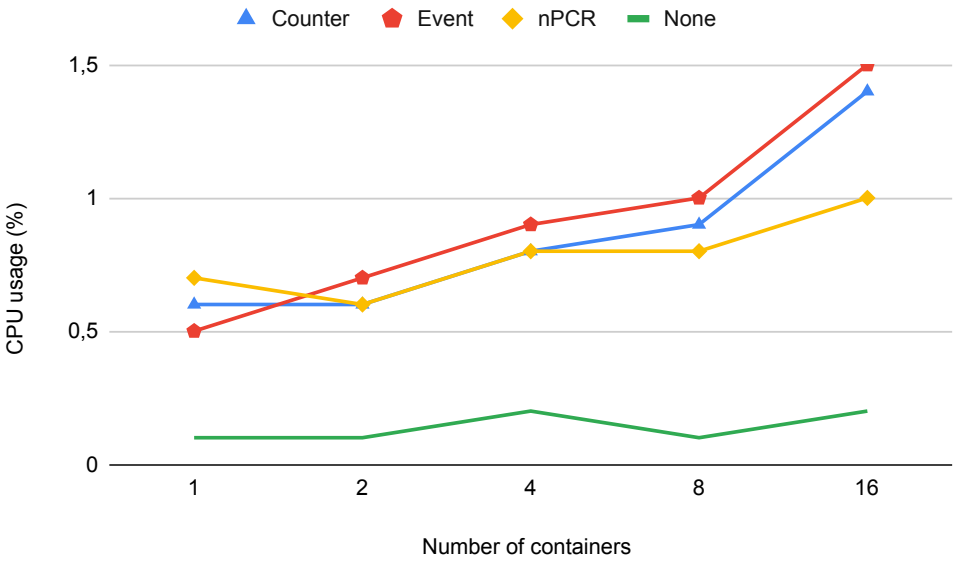

Figure 9.3. Percentages of the CPU usage in the attesting platform

that validating this list will authenticate all containers in operation on the system. Since all the solution relies on Keylime [32] for the attestation process is possible to perform comparisons. The chart in Figure 9.5 compares the attestation latency, the TPM quote time is omitted because is common to all solutions. The TPM quote generation is more than one second so is the most time-consuming operation. The "TORSEC" solution is a little faster in the verification process but considering the TPM quote generation the difference is around ten milliseconds on an operation that takes more than one second so the difference is negligible.

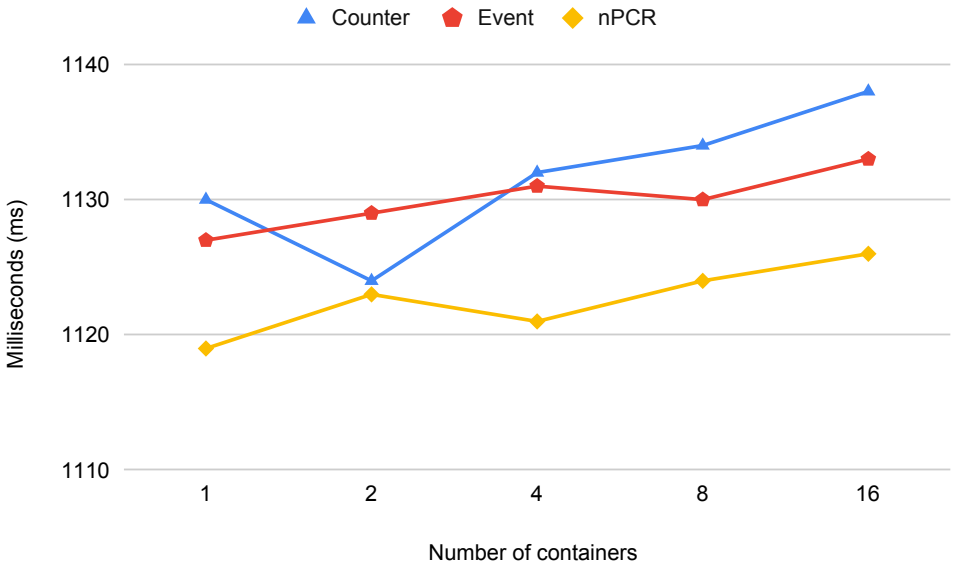

Figure 9.4. Comparison on the attestation latency in milliseconds (ms)

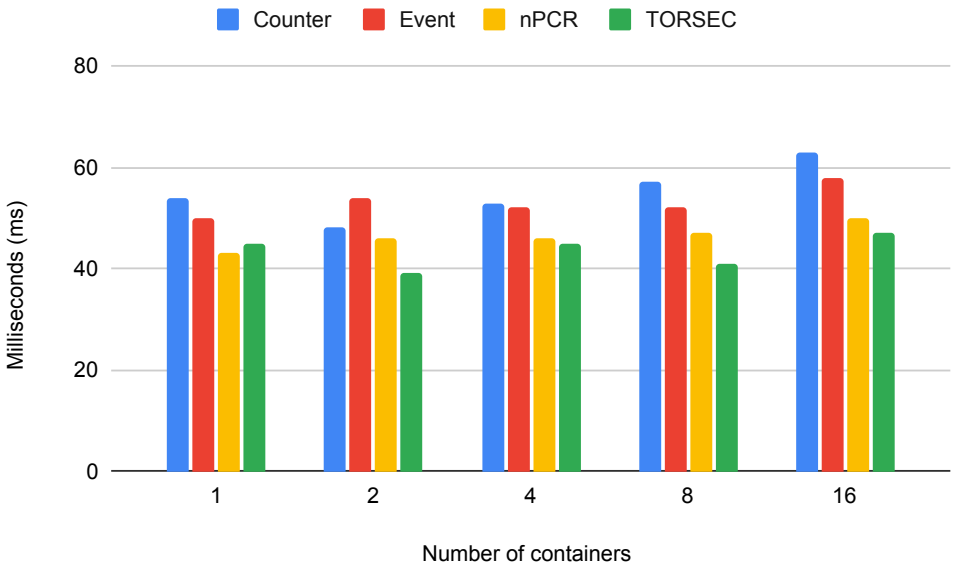

Figure 9.5. Comparison on the attestation latency in milliseconds (ms)

Chapter 10

Conclusions and future work

The main aim of this thesis was to present a viable approach for overseeing the integrity of containers, a lightweight virtualization technique. This objective has been accomplished through a series of distinct proposals, each having its own strengths and weaknesses. All of these proposals share a common foundation: the development of novel IMA templates that extend the capabilities of the IMA measurement list, enabling a verifier to discern the source namespace responsible for a given entry. To validate the effectiveness of these proposals, they were put to the test using Keylime, a framework designed for periodic remote attestation of physical platforms, utilizing TPM 2.0 as a hardware Root of Trust. Modifications were made to the Keylime code to seamlessly integrate the new template, enabling the verifier to interpret the new template and support the attestation of individual containers.

Another significant advancement made to facilitate container verification involves enabling the association between a container's ID and the IMA namespace identifier at the user level. This was achieved by introducing a new entry in the proc filesystem that provides details regarding the IMA namespace identifier of a specific process. Consequently, by identifying the primary process of a container, it becomes feasible to determine its corresponding IMA namespace identifier. Furthermore, due to the hierarchical arrangement of user namespaces, it becomes feasible to ascertain any additional namespaces created by the container.

The research conducted in this thesis offers a method for conducting container attestation by leveraging distinct IMAs for each namespace. The devised solution exhibits significant scalability and is adaptable to various containerization technologies. With the suggested Keylime solution, the process takes roughly 1.2 seconds, as demonstrated in the tests (Section 9). Additionally, it becomes evident that nearly 90% of the attestation time is dedicated to generating the TPM 2.0 quote. Consequently, to enhance attestation speed, there is a need to enhance the performance of the TPM 2.0 chip or investigate alternative technologies for use as a hardware Root of Trust

The suggested solutions serve as valuable initial steps toward achieving container attestation regardless of the containerization technology in use. Additional solutions have been put forth due to the absence of a standardized IMA namespace implementation. Once a definitive standard is established, these diverse solutions can serve as potential starting points, allowing for the selection of the most suitable approach based on other design considerations.

10.1 Possible enhancements

This solution can be enhanced with some additions. The implementation of many of them depends on the implementation of the IMA namespace as a standalone namespace. When the IMA namespace becomes a fully functional namespace, so no more included inside the user namespace. Some problems will be overcome, like the lazy init of the IMA namespace. Now for compatibility reasons is needed to initialize the IMA namespace after the creation of the user namespace. The print of a 1 in the active file.

10.1.1 Whitelist generation

Developing a mechanism to generate a whitelist for a container based on its image is an operation that demands significant computational resources. Nonetheless, it is indispensable for facilitating streamlined verification, ensuring the container exclusively runs programs associated with the given image. This procedure is especially beneficial since the majority of containers are designed to offer a service.

The program execution sequence within the container and their corresponding hash values constitute the components of this predictive list. While creating this list is relatively straightforward for containers with straightforward functionalities designed to start and provide services, it becomes more intricate for complex or interactive containers.

10.1.2 Docker save the measurement log

To execute the nPCR solution, it's essential to retain the namespace list before closing it. However, the present kernel implementation encourages interaction solely with dedicated filesystems, making file management at the kernel level undesirable. The existing approach offloads the task of gathering the list to a script executed during container startup. While this grants users more control, it also places added responsibility on them; neglecting to run the script results in a failure to collect the list, thereby hindering container attestation.

Addressing this issue involves incorporating a mechanism to save the list upon the container's closure. Achieving this necessitates integration with the Docker system. However, to ensure a detailed understanding of the container's state without granting excessive permissions to the user, this approach stands as the sole viable option.

10.1.3 Flooding checks

An issue arises regarding containers being capable of conducting measurements and transmitting them to the host list. Maliciously executed, this action could lead to an inundation of the host's list. Given that the list resides in kernel memory, there exists the potential for of saturation the kernel memory resources.

This concern is also applicable to the conventional IMA implementation, though it necessitates a malicious process incessantly generating files to be measured, subsequently overloading the host's list. However, in the IMA setup, the host list is structured using a hash table, rendering complete memory saturation challenging as the system eventually encompasses measurements for all commonly used programs. Such an assumption cannot be extended to container attestation, as the entries might not converge due to the continual creation of new containers.

To surmount this challenge, it becomes imperative to evaluate the number of measurements performed by the container and compare it with the anticipated count of measurements expected from the container. This assessment can be contextual, dependent on the container's phase. It is reasonable to expect a surge in measurements when a container is newly initiated; however, as the necessary programs are progressively measured, a lower measurement ratio is generally anticipated.

10.1.4 Obfuscation of hashes in nPCR solution

An improvement for the nPCR solution described in Section 7.4 could involve the utilization of a unique key for each namespace. This key would play a role during PCR extension, serving to obscure the current nPCR value. This approach has the potential to obscure nPCR values from different containers when presented to a verifying entity. Given that the host list encompasses all nPCR values across the system, this enhancement enables the verifier to focus solely on validating whether the simulated extension-derived nPCR value aligns with the one stored in the list. This validation can be achieved by conducting an additional hash computation using both the secret and the value acquired through the simulated extension.

The challenge lies in effectively transmitting the secret to the verifier. A further complication within the present setup involves granting verifiers the authority to attest containers. Presently, all verifiers have the capability to request verification for any container without constraint. This unrestricted access lacks a system for associating specific containers with authorized verifiers. The proposed solution aims to establish a functional key exchange process that furnishes verifiers with container-specific keys. By implementing this approach, verifiers can acquire container secrets, thwarting unauthorized verification attempts by unregistered containers. Introducing a comprehensive authentication and authorization framework for verifiers with respect to individual containers promises to resolve these concerns. To achieve this, the implementation necessitates the integration of verifier registration, authentication protocols, and a mechanism for secure key exchange.

Bibliography

[1] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation", International Journal of Information Security, vol. 10, April-June 2011, pp. 63–81, DOI 10.1007/s10207-011-0124-7

[2] C. Mitchell, "Trusted computing", Institution of Engineering and Technology, November 2005, ISBN: 978-0-863-41525-8

[3] J. M. Rushby, "Design and verification of secure systems", Proceedings of the eighth symposium on Operating systems principles - SOSP '81, Pacific Grove (California, USA), December 14-16, 1981, pp. 12–21, DOI 10.1145/800216.806586

[4] P. Bjesse, "What is formal verification?", ACM SIGDA Newsletter, vol. 35, December 2005, pp. 1–es, DOI 10.1145/1113792.1113794

[5] IBM, "What is an attack surface?", https://www.ibm.com/topics/attack-surface

[6] Trusted Computing Group, "Official tcg website", https://trustedcomputinggroup.org/

[7] S. Kinney, "Trusted platform module basics", Elsevier, July 2006, ISBN: 978-0-7506-7960-2

[8] D. Feng, Y. Qin, X. Chu, and S. Zhao, "Remote attestation", Trusted Computing (W. D. G. GmbH, ed.), pp. 197–242, De Gruyter, 2017, DOI 10.1515/9783110477597-007

[9] S. Kinney, "Platform configuration registers", Trusted Platform Module Basics (E. Science, ed.), pp. 53–64, Newnes, 2006, DOI 10.1016/b978-075067960-2/50007-5

[10] W. Arthur, D. Challener, and K. Goldman, "A practical guide to TPM 2.0", Apress, January 2015, ISBN: 978-1-4302-6584-9

[11] F. Bohling, T. Mueller, M. Eckel, and J. Lindemann, "Subverting linux's integrity measurement architecture", Proceedings of the 15th International Conference on Availability, Reliability and Security, Virtual Event (Ireland), August 25-28, 2020, pp. 1–10, DOI 10.1145/3407023.3407058

[12] T. Sugandhi, "Ima: support for measuring kernel integrity", https://lore.kernel.org/linux-security-module/27bce16a-9e95-2559-af37-b47b81bdcd2e@linux.microsoft.com/T/

[13] W. Luo, Q. Shen, Y. Xia, and Z. Wu, "Container-ima: A privacy-preserving integrity measurement architecture for containers", 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), Beijing (China), September 23-25, 2019, pp. 487–500

[14] R. Xiaoyu, "Boot with integrity, or don't boot", Platform Embedded Security Technology Revealed (P. Hauke, C. Collins, M. Maldonado, and S. Weiss, eds.), pp. 143–163, Apress, 2014, DOI 10.1007/978-1-4302-6572-6_6

[15] Linux Manual Page, "Unshare command", https://www.man7.org/linux/man-pages/man1/unshare.1.html

[16] Docker Inc., "Install docker engine on ubuntu", https://docs.docker.com/engine/install/ubuntu/

[17] Opencontainers, "Runc command", https://github.com/opencontainers/runc

[18] Kubernetes, "Kubernetes, concepts overview", http://kubernetes.io/

[19] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu, and T. Jaeger, "Security namespace: making linux security frameworks available to containers", 27th USENIX Security Symposium (USENIX Security 18), Baltimore (Maryland, USA), August 15-17, 2018, pp. 1423–1439

[20] S. Berger, "Linux ima namespaces", https://lore.kernel.org/lkml/20220201203735.164593-1-stefanb@linux.ibm.com/

[21] Linux Manual Page, "Linux namespaces", https://man7.org/linux/man-pages/man7/namespaces.7.html

[22] Linux Manual Page, "Linux mount namespace", https://man7.org/linux/man-pages/man7/mount_namespaces.7.html
[23] Linux Manual Page, "Linux network namespace", https://man7.org/linux/man-pages/man7/network_namespaces.7.html
[24] Linux Manual Page, "Linux ipc namespace", https://man7.org/linux/man-pages/man7/ipc_namespaces.7.html
[25] Linux Manual Page, "Linux uts namespace", https://man7.org/linux/man-pages/man7/uts_namespaces.7.html
[26] Linux Manual Page, "Linux user namespace", https://www.man7.org/linux/man-pages/man7/user_namespaces.7.html
[27] Linux Manual Page, "Linux cgroups", https://man7.org/linux/man-pages/man7/cgroups.7.html
[28] Linux Manual Page, "Linux cgroup namespace", https://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html
[29] Linux Manual Page, "Linux time namespace", https://man7.org/linux/man-pages/man7/time_namespaces.7.html
[30] Busybox Project, "About busybox", https://www.busybox.net/about.html
[31] N. Schear, P. T. Cable, T. M. Moyer, B. Richard, and R. Rudd, "Bootstrapping and maintaining trust in the cloud", Proceedings of the 32nd Annual Conference on Computer Security Applications, Los Angeles (California, USA), December 05-08, 2016, pp. 65–67, DOI 10.1145/2991079.2991104
[32] Keylime Project, "Official documentation", https://keylime.readthedocs.io/en/latest/
[33] The Linux Kernel, "Using the initial ram disk (initrd)", https://www.busybox.net/about.html
[34] T. Bowden, B. Bauer, J. Nerin, S. Feng, and S. Seibold, "Proc filesystem documentation", https://kernel.org/doc/Documentation/filesystems/proc.txt
[35] M. Eder, "Hypervisor- vs. Container-based Virtualization", Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM), Munich (Germany), July 25-26, 2016, pp. 1–7, DOI 10.2313/NET-2016-07-1_01
[36] M. Souppaya, J. Morello, and K. Scarfone, "Application container security guide", NIST SP800-190, September 2017, DOI 10.6028/NIST.SP.800-190
[37] Docker Inc., "Overview", https://docs.docker.com/get-started/overview/
[38] S. Malladi, J. Alves-Foss, and R. B. Heckendorn, "On preventing replay attacks on security protocols", Proc. International Conference on Security and Management, Moscow (Idaho, USA), June 15, 2002, pp. 1–8
[39] Docker Inc., "Isolate containers with a user namespace", https://docs.docker.com/engine/security/userns-remap/
[40] P. D. Tender, "Basic docker commands", Building and Deploying Container Workloads on Azure (P. D. Tender, ed.), pp. 38–59, Apress, 2021, DOI 10.1007/978-1-4842-7807-9_2
[41] G. Kroah-Hartman, "Things You Never Should Do in the Kernel", Linux Journal, vol. 1, April 2005, p. 1
[42] Linux Manual Page, "Dmesg and linux kernel logging", https://man7.org/linux/man-pages/man1/dmesg.1.html
[43] Linux Manual Page, "Signals", https://man7.org/linux/man-pages/man7/signal.7.html
[44] Linux Manual Page, "Trap signals", https://www.man7.org/linux/man-pages/man1/trap.1p.html
[45] Docker Inc., "Dockerfile reference", https://docs.docker.com/engine/reference/builder/
[46] Docker Inc., "Docker build", https://docs.docker.com/engine/reference/commandline/build/
[47] Postman Project, "Official website", https://www.postman.com/product/what-is-postman/
[48] Git Project, "Official documentation", https://www.git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F

[49] User Mode Linux Core Team, "User mode linux howto", https://www.kernel.org/doc/html/v5.9/virt/uml/user_mode_linux.html
[50] TPM2 Developer Community, "Tpm software stack 2.0 tcg spec compliant implementation", https://tpm2-tss.readthedocs.io/en/latest/
[51] TPM2 Developer Community, "Documentation tpm-tools", https://tpm2-tools.readthedocs.io/en/latest/
[52] Keylime Project, "Github page", https://github.com/keylime/keylime
[53] Docker Inc., "Docker rootless setup", https://docs.docker.com/engine/security/rootless/
[54] Docker Inc., "Container namespace setup", https://docs.docker.com/engine/security/userns-remap/

Appendix A

User's manual

A.1 Linux installation

To install the solution, start by installing Git, then clone the repository and change the current directory into it. The following code works on a Debian-like system, to install it on other distributions check the package manager used on it.

```
$ sudo apt install git
$ git clone git@github.com:stefanberger/linux-ima-namespaces.git
$ cd linux-ima-namespaces
```

Git [48], a distributed version control system, is a fast and efficient tool specifically created to manage projects of various sizes, ranging from small to extremely large ones. It is freely available as an open-source software.

A.1.1 Compilation

Install the dependency using:

```
$ sudo apt-get install git fakeroot build-essential libncurses-dev \
  xz-utils libssl-dev bc flex libelf-dev bison
```

The cited packages are used for:

- `fakeroot`: executes a command within an environment that simulates root privileges for performing file manipulation. Operates by substituting the file manipulation library functions (such as `chmod` etc.) with counterparts that replicate the intended effects of the genuine library functions if the user possesses root privileges.
- `build-essential`: specific to Debian, it comprises a collection of essential packages necessary for creating a Debian package (deb). These packages include `libc, gcc, g++, make, dpkg-dev`, and others. By installing the build-essential package, you automatically install all these required packages as dependencies with a single command.
- `ncurses-dev`: is a programming library that offers an Application Programming Interface (API) enabling developers to create Text-basedUser Interfaces (TUI) in a manner independent of the terminal. It serves as a toolkit for building application software with `GUI-like` functionalities that can be executed within a terminal emulator. Additionally, it optimizes screen updates to minimize latency when utilizing remote shells.

- `xz-utils`: is a freely available data compression software that boasts a remarkable compression ratio and is suitable for various general-purpose compression tasks.
- `libssl-dev`: is a package that provides the development files and headers necessary for programming with OpenSSL, a widely used open-source cryptographic library. It allows developers to incorporate secure socket layer (`SSL`) and transport layer security (`TLS`) protocols, as well as various encryption and decryption functionalities, into their applications.
- `bc`: is an interactive algebraic language that provides arbitrary precision calculations. It adheres to the `POSIX 1003.2` draft standard and offers additional features such as multi-character variable names, an `else` statement, and support for comprehensive Boolean expressions.
- `flex`: is a software utility used for generating scanners, which are programs designed to identify lexical patterns within text. It processes input files that contain a description of the desired scanner to be generated. This description consists of pairs of regular expressions and accompanying C code, known as rules.
- `libelf-dev`: is a development library that provides the necessary files and headers for programming with ELF (Executable and Linkable Format) files. ELF is a common file format used in Linux and many other Unix-like operating systems for executables, object code, shared libraries, and core dumps.
- `bison`: is a software development tool that assists in generating parsers. Developed as part of the GNU Project, Bison specializes in creating LALR parsers based on formal grammar specifications. By providing a grammar file, typically written in a language resembling Extended Backus-Naur Form (EBNF), Bison generates `C` or `C++` code that establishes a parser capable of interpreting the designated grammar.

To compile the solution, you have two options: installing the kernel directly on the machine or using the user-mode approach A.1.2.

For installing on the bare machine, firstly copy the machine `.config` file (`$ cp -v /boot/config-$(uname -r).config`)and then modify it `$ make menuconfig`, on the menu opened you need to enable (if not already active):

- Security Options → Enable different security models;
- Security Options → Enable the securityfs filesystem;
- Security Options → Integrity Subsystem;
- Security Options → Integrity Measurement Architecture (IMA);
- Security Options → Enable multiple writes on IMA policy (if necessary to modify the policy at runtime);
- All the entries on the path General Setup → Namespaces support.

```
$ cp -v /boot/config-$(uname -r) .config
$ make menuconfig
$ make
$ make modules
$ sudo make modules_install
$ sudo make install
```

After this command the kernel and kernel modules are installed, and the entry has been added to the bootloader so restarting the computer the new kernel will work.

A.1.2 User-mode

User Mode Linux (UML) [49] offers a range of benefits and features that enhance its usability. For instance, in the event of a UML crash, the host kernel remains unaffected and continues to function properly. Moreover, UML allows running a user-mode kernel as a non-root user, providing greater flexibility and security. Debugging User Mode Linux is as straightforward as debugging any other regular process, simplifying the troubleshooting process. Additionally, UML supports profiling tools such as `gprof` and coverage testing tools like /codegcov. This enables developers to optimize performance and assess code coverage effectively.

UML serves as a safe environment for experimenting with kernel configurations without the risk of system instability or disruption. It can be used as a sandbox for testing new applications, ensuring they function as intended without impacting the host system. Developers can also evaluate new development kernels within UML, allowing them to assess their features and compatibility before deploying them in production environments. Lastly, UML enables the concurrent execution of different Linux distributions, facilitating side-by-side comparisons and compatibility testing.

User Mode Linux (UML) Kernel operates by communicating with a real Linux kernel (referred to as the host kernel). Instead of directly interacting with the hardware, UML interacts with the host kernel as if it were any other program. This enables programs to run within UML as if they were running on a regular kernel.

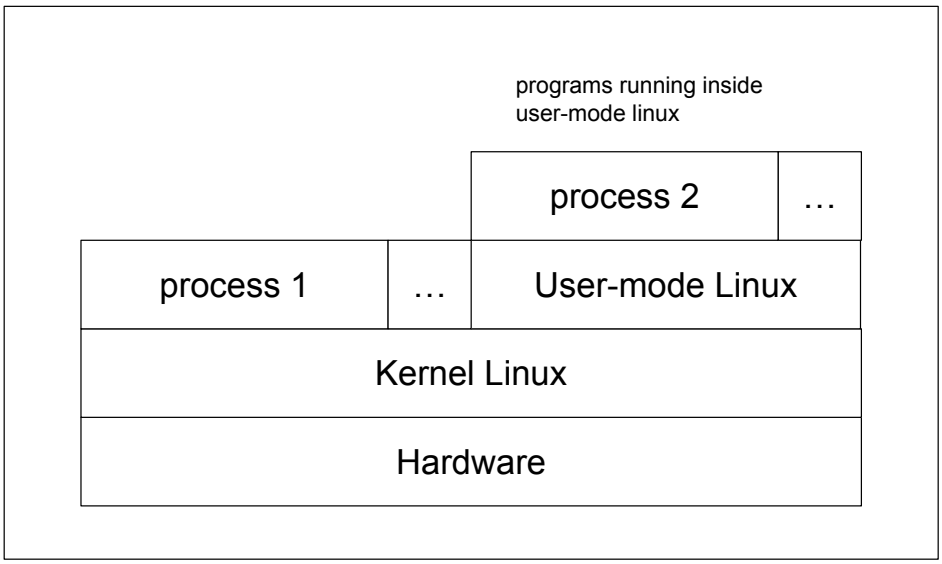

Figure A.1. UML summary schema

To compile a kernel in user mode run the following commands (in the /codemenuconfig to enable the same entry as A.1.1

```
$ make ARCH=um
$ make modules ARCH=um
$ make modules_install ARCH=um
```

Modules may also be installed directly on the machine by doing, the mount point of /coderoot_fs can be obtained by using:

```
$ sudo mount | grep -i "on / "
$ sudo mount /dev/XXXXX mnt -o loop
```

```
$ make modules_install INSTALL_MOD_PATH=‘pwd‘/mnt ARCH=um
$ sudo umount /mnt
$ make
$ sudo make install
```

To use a filesystem file we can create it using dd and create a minimal filesystem using /codede-bootstrap:

```
$ dd if=/dev/zero of=disk_image_name bs=1 count=1 seek=16G
$ mkfs.ext4 ./disk_image_name
$ sudo mount ./disk_image_name /mnt
$ sudo apt install debootstrap
$ sudo debootstrap bullseye /mnt http://deb.debian.org/debian
$ sudo chroot /mnt
$ passwd
$ exit
```

Run the user-mode kernel with the command:

```
$ ./linux mem=2048M umid=TEST ima_policy=tcb ubd0=disk_image_name \
  vec0:transport=tap,ifname=tap0,depth=128,gro=1 root=/dev/ubda \
  con=null con0=null,fd:2 con1=fd:0,fd:1
```

We can access data in the host filesystem by mounting it:

```
$ cat /proc/filesystems
$ mount none /mnt/host -t hostfs
$ mount none /mnt/home -t hostfs -o /home
```

Instead of creating a filesystem file we can use the host filesystem by loop mounting it:

```
$ sudo mount root_fs uml_root_dir -o loop
# change the entry  / to look like
  /dev/ubd/0   /   hostfs   defaults   1   1
$ find . -uid 0 -exec chown jdike {} \;
# run the user-mode with the parameter ubd0=/path/to/uml/root/directory
```

If gives the problem of “read-only filesystem” remount the filesystem in “read-write” mode with the command:

```
$ mount -o remount,rw name_fs_to_remount
```

In the case in the current host the hostfs is not configured in the kernel is possible to recompile the kernel setting the `Host filesystem` or compile it as a module setting the option `Host filesystem` option under `arch/um/fs/hostfs/hostfs.o` install it in `/lib/modules/$(uname -r)/fs` and run `insmod hostfs`

A.2 Keylime Verifier and Registrar installation

To correctly install [32] and set the verifier, is needed to clone the Keylime repository (requires git [48] installed, use `$ sudo apt install git` in a Debian environment to install it) and change the current directory into it, using the following command.

```
$ git clone https://github.com/keylime/keylime.git
$ cd keylime
```

To run Keylime, you must have /codePython 3.6 or a newer version installed (use `$ sudo apt install python3.10` in a Debian environment to install it). For installation, you can utilize the automated shell script, `installer.sh`. The script offers several command-line options, including:

```
Usage: ./installer.sh [option...]
Options:
-k              Download Keylime (stub installer mode)
-m              Use modern TPM 2.0 libraries; this is the default
-s              Install & use a Software TPM emulator (development only)
-p PATH         Use PATH as Keylime path
-h              This help info
```

Example of usage:

```
$ sudo ./installer.sh -k
```

The installation script already includes the necessary dependencies, which are:

- **all the required python packages**;
- **tpm-tss** [50]: offers a standardized API for accessing TPM functions. By utilizing this specification, application developers can create interoperable client applications designed for enhanced tamper-resistant computing Can be installed separately by using:

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

- **tpm2-tools** [51]: comprise a set of command-line tools, both low-level and aggregate, that grant access to a tpm2.0 compatible device. These tools are built on the `tpm-tss` library. Can be installed separately by using:

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

A.2.1 Configuring basic (m)TLS setup

Keylime employs `mTLS` authentication between its various components. Upon initial startup, the verifier generates a CA (Certificate Authority) for this purpose, which is stored under `/var/lib/keylime/cv_ca/`. Within this directory, you will find files pertaining to three distinct components. Keylime permits each component to utilize its unique server and client keys, along with a roster of trusted certificates for establishing `mTLS` connections

- **root CA**: `cacert.crt` contains the root CA certificate, this needs also to be deployed on the agent, otherwise it will be unverifiable by the tenant;
- **server certificate and key**: `server-cert.crt` and `server-{private,public}.pem` are utilized by the registrar and verifier for their HTTPS interfaces;
- **client certificate and key**: `client-cert.crt` and `client-{private,public}.pem`, the tenant employs these key and certificate pairs to authenticate against the verifier, registrar, and agent. Additionally, the verifier uses its key and certificate to authenticate against the agent.

A.2.2 Manual installation

The first thing that needs to be done is to clone the Keylime repository, this step is common to the standard installation, move to the Keylime directory, and install the script.

```
$ git clone https://github.com/keylime/keylime.git
$ cd keylime
$ sudo pip3 install . -r requirements.txt
```

The configuration file needs to be copied.

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

Add TPM 2.0 Resource manager.

```
$ sudo useradd --system --user-group tss
```

TPM Command Transmission Interface (TCTI) configuration.

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

Start TPM 2.0 Resource Manager service.

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

A.3 Keylime Agent installation

This is a Rust implementation of the Keylime agent so firstly rust and its dependencies need to be installed using the following commands:

```
# on Debian-like distributions dependencies
$ sudo apt-get install clang libarchive-dev libssl-dev \
  libtss-dev libzmq3-dev curl

# on Fedora dependencies
$ sudo dnf install clang libarchive-devel openssl-devel \
  tpm2-tss-devel zeromq-devel curl

# Rust installation
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Clone the repository and change the current directory:

```
$ git clone https://github.com/keylime/rust-keylime.git
$ cd rust-keylime
```

Logging env has to be set using:

```
$ RUST_LOG=keylime_agent=trace cargo run --bin keylime_agent
```

Install the executables and the unit file then start the service:

```
$ make
$ sudo make install
$ sudo systemctl start keylime_agent
```

Building package with cargo-deb:

```
# Install cargo-deb
$ rustup update
$ cargo install cargo-deb

# Build Debian package
$ cargo deb -p keylime_agent
```

A.4 Ansible installation

To install and set up Keylime, you have the option to deploy Keylime alongside the Keylime Rust agent. This particular role is designed for deploying Keylime in conjunction with a Hardware TPM. However, if you prefer to deploy Keylime with a software TPM emulator for development purposes or to gain initial familiarity, it is recommended to use the Ansible Keylime Soft TPM role instead.

- **Hardware TPM**: clone Ansible Keylime using the following command. After cloning, navigate to the newly created directory.

  ```
  $ git clone https://github.com/keylime/ansible-keylime.git
  $ cd ansible-keylime
  ```

  Execute the example playbook against your designated remote host(s)

  ```
  $ ansible-playbook -i your_hosts playbook.yml
  ```

  Now is possible to run the services: verifier, registrar, agent:

  ```
  $ keylime_verifier
  $ keylime_registrar
  $ RUST_LOG=keylime_agent=trace cargo run --bin keylime_agent
  ```

- **TPM emulator**: this solution is exclusively intended for testing purposes and should not be employed in a production environment. To proceed, clone the repository using the provided method and navigate to the newly created directory

  ```
  $ git clone https://github.com/keylime/
    keylime-vagrant-ansible-tpm-emulator.git
  $ cd keylime-vagrant-ansible-tpm-emulator
  ```

The Vagrantfile facilitates the provisioning of virtual machines for local testing. To execute it with additional arguments, add the specified arguments to the Vagrant command:

- `--instances`: can specify the number of Keylime virtual machines to create. If no value is provided, the default number is set to `1`;
- `--repo`: is designed to facilitate Keylime development. It establishes a connection between the local Keylime Git repository and the virtual machine, enabling you to test your code within the VM environment. This process is optional and involves mounting the specified repository directory at `/root/keylime-dev`;
- `--cpus`: the number of CPUs used. Defaults to `2`;
- `--memory`: the amount of memory to assign. Defaults to `2048`;
- `--qualityoflife`: includes some additional features, such as enhancing the bash shell prompt with `Powerline` and introducing an alias for the `ls` command (`ll` for `ls -la`).

A virtualization provider is needed, for example, VirtualBox, in case not already installed, can be installed on a Debian-like system using the following commands:

```
$ sudo apt-get install virtualbox
$ sudo apt-get install virtualbox-ext-pack
```

Here's an example of a deployment using VirtualBox as the virtualization provider:

```
$ vagrant --instances=2 --repo=/home/user/keylime --cpus=4 \
  --memory=4096  up --provider virtualbox --provision
```

Also, other virtualization providers could be used, for example, `libvirt`:

```
$ vagrant --instances=2 --repo=/home/user/keylime --cpus=4 \
  --memory=4096  up --provider libvirt --provision
```

To personalize these defaults without the need to repeatedly specify them on the command line, you can utilize a `vagrant_variables.yml` file. The easiest method is to duplicate `vagrant_variables.yml.sample` and then edit the duplicated `vagrant_variables.yml` file to suit your preferences

```
$ cp vagrant_variables.yml.sample vagrant_variables.yml
```

Although the defaults can be customized in `vagrant_variables.yml`, you still have the option to override them using command line options. After starting the virtual machine, utilize `vagrant ssh` to SSH into the VM and then execute `$ sudo su -` to become root. The TPM emulator will be operational.

Before successfully starting `keylime_agent`, it's highly likely that you'll need to export the appropriate `TPM2TOOLS_TCTI` environment variable. To achieve this, use the following command:

```
$ export TPM2TOOLS_TCTI="mssim:port=2321"
```

You may initiate the various components using the following commands:

```
$ keylime_verifier
$ keylime_registrar
$ keylime_agent
$ keylime_node
```

A.5 Usage

A.5.1 Command line commands

After the setup, some commands become available for use from the command line:

- `keylime_verifier`: start the verifier service;
- `keylime_tenant`: start the tenant service;
- `keylime_userdata_encrypt`: encryption of a given file ;
- `keylime_registrar`: start the registrar service;
- `keylime_ca`: handle the certification authority;
- `keylime_attest`: verification of the state of all the agents registered in the persistence storage;
- `keylime_convert_runtime_policy`: for the runtime policy conversion;
- `keylime_sign_runtime_policy`: signs Keylime runtime policies using `DSSE`;
- `keylime_upgrade_config`: parses the content of a configuration file and uses the data to replace the values in templates to generate new configuration files;
- `keylime_create_policy`: create a JSON allowlist/policy from given files as input;
- `Keylime_verify_container`: the newly inserted API that handles the verification of a container
- `keylime_agent`: start the agent service.

A.5.2 Example of usage

Illustration of the setup and application of "Runtime Integrity Monitoring", wherein it is essential to ensure the proper configuration of IMA 3.1.

A.5.3 Keylime runtime policies

In its simplest form, a runtime policy consists of "golden" cryptographic hashes representing files' unaltered states or keys that can be loaded onto keyrings for IMA verification. Keylime will load the runtime policy into the Keylime Verifier. The Verifier will then regularly retrieve TPM quotes from PCR 10 on the agents' TPM and cross-validate the agents' file(s) state with the specified policy. If any object has been tampered with or an unexpected key has been loaded onto a keyring, the hashes will not match, leading Keylime to categorize the agent as failed. Additionally, if any files trigger actions outlined in the ima-policy that are not listed in the allowlist, Keylime will also classify the agent as failed.

A runtime policy can be generated in an easy way using the script `create_runtime_policy.sh`

```
Usage: $0 -o/--output_file FILENAME [-a/--algo ALGO]
[-x/--ramdisk-location PATH] [-y/--boot_aggregate-location PATH]
[-z/--rootfs-location PATH] [-e/--exclude_list FILENAME]
[-s/--skip-path PATH]"

optional arguments:
-a/--algo                    (checksum algorithmi to be used,
                            default: sha1sum)
-x/--ramdisk-location        (path to initramdisk, default:
```

```
                            /boot/, set to "none" to skip)
-y/--boot_aggregate-location (path for IMA log, used for boot
                            aggregate extraction, default:
                            /sys/kernel/security
                            /ima/ascii_runtime_measurements,
                            set to "none" to skip)
-z/--rootfs-location         (path to root filesystem, default: /,
                            cannot be skipped)
-e/--exclude_list            (filename containing a list of paths
                            to be excluded (i.e., verifier will
                            not try to match checksums),
                            default: none)
-s/--skip-path               (comma-separated path list, files found
                            there will not have checksums calculated,
                            default: none)
-h/--help                     show this message and exit
```

The output file, `runtime_policy_keylime.json` is readily applicable to `keylime_tenant` using the `--runtime-policy` option.

Creation of more complex policies

The second script offers users the ability to construct more intricate policies through various options, including keyring verification, and IMA verification keys, generating allowlists from IMA measurement logs, and extending existing policies.

```
usage: keylime_create_policy [-h] [-B BASE_POLICY] [-k] [-b]
[-a ALLOWLIST] [-m IMA_MEASUREMENT_LIST] [-i IGNORED_KEYRINGS]
[-o OUTPUT] [--no-hashes] [-A IMA_SIGNATURE_KEYS]

options:
-h, --help            show this help message and exit
-B BASE_POLICY, --base-policy BASE_POLICY
                      Merge new data into the given JSON
                      runtime policy
-k, --keyrings        Create keyrings policy entries
-b, --ima-buf         Process ima-buf entries other than those
                      related to keyrings
-a ALLOWLIST, --allowlist ALLOWLIST
                      Use given plain-text allowlist
-m IMA_MEASUREMENT_LIST, --ima-measurement-list IMA_MEASUREMENT_LIST
                      Use given IMA measurement list for keyrings
                      and critical data extraction rather than
                      /sys/kernel/security/ima/ascii_runtime_measurements
-i IGNORED_KEYRINGS, --ignored-keyrings IGNORED_KEYRINGS
                      Ignored the given keyring; this option may be
                      passed multiple times
-o OUTPUT, --output OUTPUT
                      File to write JSON policy into; default is
                      to print to stdout
--no-hashes           Do not add any hashes to the policy
-A IMA_SIGNATURE_KEYS, --add-ima-signature-verification-key
                      IMA_SIGNATURE_KEYS
                      Add the given IMA signature verification key to
                      the Keylime-internal 'tenant_keyring'; the key
                      should be an x509 certificate in DER or PEM format
                      but may also be a public or private key
                      file; this option may be passed multiple times
```

A.5.4 Remotely provisioning agents

With the runtime policy now accessible, we can proceed to send it to the verifier using the `keylime_tenant` command.

```
$ touch payload  # create empty payload for example purposes
$ keylime_tenant -c add --uuid <agent-uuid> -f payload --runtime-policy \
  /path/to/policy.json
  # if the agent is already registered use "-c update"
```

A.5.5 Test the configuration

Develop a script that performs an action that is not listed in your runtime policy. Execute this script with root privileges on the agent machine. Subsequently, you will observe the following output on the verifier, indicating a change in the agent status to `failed`.

```
keylime.tpm - INFO - Checking IMA measurement list...
keylime.ima - WARNING - File not found in allowlist:
  /root/evil_script.sh
keylime.ima - ERROR - IMA ERRORS: template-hash 0 fnf
  1 hash 0 good 781
keylime.cloudverifier - WARNING - agent D432CD44-D2F1
  -4A97-9EE7-75BD41D80000 failed, stopping polling
```

A.5.6 Main APIs

Brief description of the most important APIs [52] for each component of the system

- **Verifier**
  - **GET** on `/v2.1/agents/{agent_id:UUID}`: get status of agent `agent_id` from the verifier;
  - **POST** on `/v2.1/agents/{agent_id:UUID}`: add new agent `instance_id` to the verifier.
- **Agent**
  - **GET** on `/v2.1/keys/pubkey`: retrieves agents public key;
  - **GET** on `/v2.1/quotes/integrity`: get integrity quote from node;
  - **GET** on `/v2.1/quotes/identity`: get identity quote from node.
- **Registrar**
  - **GET** on `/v2.1/agents/`: get ordered list of registered agents;
  - **GET** on `/v2.1/agents/{agent_id:UUID}`: get EK certificate, AIK, and optional contact ip and port of agent `agent_id`;
  - **POST** on `/v2.1/agents/{agent_id:UUID}`: add agent `agent_id` to registrar.

A.6 Docker Installation

The subsequent installation and configuration instructions are intended for systems running Ubuntu Focal 20.04 or later iterations.

A.6.1 Unistall old versions

Make certain that there are no remnants of previous Docker installations [16] on the system using these commands

```
$ for pkg in docker.io docker-doc docker-compose podman-docker containerd \
  runc; do sudo apt-get remove $pkg; done

$ sudo apt remove docker-desktop

$ rm -r $HOME/.docker/desktop
$ sudo rm /usr/local/bin/com.docker.cli
$ sudo apt purge docker-desktop
```

A.6.2 Setup the repository

Following the update of the apt package manager, the official GPG key for Docker is incorporated, and the repository is established.

```
$ sudo apt-get update
$ sudo apt-get install ca-certificates curl gnupg

$ sudo install -m 0755 -d /etc/apt/keyrings
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
  sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
$ sudo chmod a+r /etc/apt/keyrings/docker.gpg

$ echo \ "deb [arch="$(dpkg --print-architecture)" \
  signed-by=/etc/apt/keyrings/docker.gpg] \
  https://download.docker.com/linux/ubuntu \
  "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable"
$ sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

$ sudo apt-get update
```

A.6.3 Install docker engine

Concluding the installation of the package required for the Docker engine sourced from the previously established repository.

```
$ sudo apt-get install docker-ce docker-ce-cli containerd.io \
  docker-buildx-plugin docker-compose-plugin
```

A.6.4 Installation of the Docker GUI

Ensure that the system has KVM support, the module can be run manually

```
$ modprobe kvm

$ modprobe kvm_intel  # Intel processors

$ modprobe kvm_amd    # AMD processors
```

Then download the docker Debian package from the Official repository and install it with the following commands

```
$ sudo apt-get update
$ sudo apt-get install ./docker-desktop-<version>-<arch>.deb
```

A.7 Configuration

A.7.1 Rootless Configuration

The rootless mode [53] permits the execution of the Docker daemon and containers by a non-root user, thereby reducing potential vulnerabilities in both the daemon and the container runtime.

Prerequisites

- installation of `dbus-user-session` package if not installed using

```
$ sudo apt-get install -y dbus-user-session
```

- overlay2 enabled, which is by default;
- 18.04 or more recent version of Ubuntu.

Installation

Disable the system-wide docker daemon it is running

```
$ sudo systemctl disable --now docker.service docker.socket
```

Install the `docker-rootless-setuptools` with the command

```
$ sudo apt-get install -y docker-ce-rootless-extras
```

Run the installation command as a non-root user to set up the daemon

```
$ dockerd-rootless-setuptool.sh install
```

A.7.2 Namespaced Configuration

The most effective approach [54] to thwart privilege-escalation attacks originating from within a container involves setting up your container's applications to operate with non-privileged user accounts. In cases where a container's processes necessitate the root user's privileges, it's possible to remap this user to a lower-privileged user on the Docker host. The remapped user is allocated a range of User IDs (UIDs) that operate within the namespace as typical UIDs ranging from `0` to `65536`, while having no privileges on the host machine.

The remapping process is managed through two files: `/etc/subuid` and `/etc/subgid`. These files function similarly, but one is responsible for defining the user ID range, while the other addresses the group ID range. Upon configuring Docker to employ the `userns-remap` functionality, you have the choice to either designate an existing user and/or group or opt for the "default" setting. Opting for "default" results in the creation and utilization of a user and group named `dockremap` for this particular purpose.

Prerequisites

- The linked subordinate UID and GID ranges should be linked to an established user, although this association functions as an implementation element. The user takes ownership of the isolated storage directories located within `/var/lib/docker/`. If there's no intention to utilize an existing user, Docker retains the ability to generate one and employ it accordingly. If the preference is to apply an existing username or user ID, it necessitates prior existence. Commonly, this involves ensuring the corresponding entries are present within `/etc/passwd` and `/etc/group`.

- For any destinations on the Docker host that require write access by the non-privileged user, it's essential to modify the permissions of those destinations accordingly. This holds true even if the intention is to utilize the `dockremap` user, which Docker generates automatically. However, the permissions adjustment should occur after configuring and subsequently restarting Docker.
- Enabling `userns-remap` operates as a concealment mechanism for pre-existing image and container layers, as well as various other Docker entities contained within `/var/lib/docker`. This adjustment arises from Docker's necessity to modify the ownership of these assets and place them in a subdirectory within `/var/lib/docker`. It's advisable to activate this functionality during the setup of a fresh Docker instance rather than an already established one. Conversely, when `userns-remap` is disabled, any resources generated during its activation become inaccessible.

Installation

You have the option to initiate `dockerd` using the `--userns-remap` flag, or you can adhere to this process to set up the daemon via the `daemon.json` configuration file.

```
$ dockerd --userns-remap="testuser:testuser"

  or

{
  "userns-remap": "testuser"
}
```

Listing A.1. Instruction for Docker namespaced configuration

A.8 Script for namespace initialization and list collection

To perform the late-init of the IMA namespace and collect the namespace's list upon container closure, you must execute a script alongside the container. Inserting this script into the Dockerfile is necessary to launch it concurrently with the container. So the Dockerfile will look like:

```
FROM ubuntu:latest
COPY script_name.sh /
RUN chmod +x /script_name.sh
ENTRYPOINT ["/bin/bash", "/script_name.sh"]
```

Listing A.2. Dockerfile example

By doing this the following script will be executed on container startup.

```
exit_proc() {
    LSENT=$(ls -la /proc/1/ns | grep ima_ns)
    NSID=$(echo "$LSENT" | cut -d "[" -f 2 | cut -d "]" -f 1)
    cat /mnt/ima/ascii_runtime_measurements > where_save_list/$NSID
    exit 0
}

trap exit_proc SIGTERM SIGINT

mount -t securityfs none /mnt
echo 1 > /mnt/ima/active

# policy insertion this is a example of policy
echo -e 'measure func=BPRM_CHECK\nmeasure func=FILE_MMAP mask=MAY_EXEC\n' >
    /mnt/ima/policy
```

```
while true; do
    sleep 10
done
```

Listing A.3. Script for save the container's list upon closure and IMA namespace setup

A.8.1 Run the container

The Dockerfile needs to be saved and built. Those commands are meant to be executed in the same folder of the Dockerfile.

```
$ docker build --tag 'image_name' .
```

Following the successful building of the Dockerfile, you can proceed to run the container. To enable the container to perform mount operations, it is necessary to grant it the `sys_admin` capability.

```
$ docker run -i -t --cap-add sys_admin -v path_to_save_list:/root
```

Appendix B

Developer's manual

B.1 Common parts for the implementation of the solutions

B.1.1 IMA namespace identifier

Insertion of the unique IMA namespace identifier (Section 7.2.2).

```
struct ima_namespace {
    unsigned long ima_ns_flags;
/* Bit numbers for above flags; use BIT() to get flag */
#define IMA_NS_LSM_UPDATE_RULES  0
#define IMA_NS_ACTIVE   1
    int id;
    struct rb_root ns_status_tree;
    rwlock_t ns_tree_lock;
    struct kmem_cache *ns_status_cache;
    ...
} __randomize_layout;
```

Listing B.1. `security/integrity/ima/ima.h`

Initialization of an IMA namespace with the identifier is an atomic operation.

```
include "ima.h"
int counter_ns = 0;
int ima_init_namespace(struct ima_namespace *ns) {
    int ret;
    ns->ns_status_tree = RB_ROOT;
    rwlock_init(&ns->ns_tree_lock);
    /* Use KMEM_CACHE for simplicity */
    ns->ns_status_cache = KMEM_CACHE(ns_status, SLAB_PANIC);
    INIT_LIST_HEAD(&ns->ima_default_rules);
    INIT_LIST_HEAD(&ns->ima_policy_rules);
    INIT_LIST_HEAD(&ns->ima_temp_rules);
    ns->ima_rules = (struct list_head __rcu *)(&ns->ima_default_rules);
    ns->ima_policy_flag = 0;
    ns->arch_policy_entry = NULL;
    ns->id = ++counter_ns;
    ...
}
```

Listing B.2. `security/integrity/ima/ima_init_ima_ns.c`

B.1.2 Proc filesystem modifications

Insertion of the entry regarding the IMA namespace identifier in the `proc` filesystem.

```
const struct proc_ns_operations imans_operations = {
    .name  = "ima_ns",
    .type  = CLONE_NEWUSER,
    .get   = imans_get,
    .put   = userns_put,
    .install = userns_install,
    .owner  = userns_owner,
    .get_parent = ns_get_owner,
};
```

Listing B.3. `security/integrity/ima/ima_fs.c`

The function used to create the entry for the `proc` filesystem.

```
static struct ns_common *imans_get(struct task_struct *task) {
    struct user_namespace *user_ns;
    int ima_id = 0;
    struct ns_common *ima_ns_common = kmalloc(sizeof(struct ns_common),
        GFP_KERNEL);
    rcu_read_lock();
    user_ns = get_user_ns(__task_cred(task)->user_ns);
    rcu_read_unlock();
    if(user_ns == NULL)
        return NULL;
    #ifdef CONFIG_IMA_NS
        if(user_ns->ima_ns != NULL)
            ima_id = ima_ns_id_from_user_ns(user_ns);
    #endif
    ima_ns_common->count = user_ns->ns.count;
    ima_ns_common->ops = user_ns->ns.ops;
    ima_ns_common->stashed = user_ns->ns.stashed;
    ima_ns_common->inum = ima_id;
    return ima_ns_common;
}
```

Listing B.4. `kernel/user_namespace.c`

B.1.3 Container identifier to IMA namespace identifier mapping

This script is capable of retrieving information about the container main process using `docker inspect`. With this information access to the `proc` filesystem, the modified version that shows the IMA namespace identifier, and retrieve it.

```
#!/bin/bash
# Return the ima_ns_id for the container
# 1 - wrong arguments
# 2 - Container not contained in the host
# 3 - Container not running or ima_ns not enabled
extract_number_from_brackets() {
    local input_string="${1#"["}"
    input_string="${input_string%"]"}"
    echo "$input_string"
}
if [ $# -ne 1 ]; then
    echo "Error: Exactly one argument is required."
```

```
    exit 1
fi
cont_id="$1"
firstchars="${cont_id:0:12}"
dockerps=$(sudo -u lo docker ps -a | grep -i "${firstchars}")
if [ "$dockerps" == "" ]; then
    echo "The container is not contained in the host"
    exit 2
fi
dockerval=$(sudo -u lo docker container inspect $1 | grep -m 1 '"Pid": ')
docker_pid=$(echo "$dockerval" | awk -F':' '{print $2}')
pid=$(echo "$docker_pid" | tr -d ' ,')
if [ "$pid" == 0 ]; then
    echo "No pid found, container may be not running or not able to create
        user namespaces."
    exit 3
fi
output=$(ls -la /proc/$pid/ns | grep -i "ima_ns")
tokens=( $output )
userns="${tokens[10]}"
OLD_IFS=$IFS
IFS=":" read -ra value <<< "$userns"
IFS=$OLD_IFS
nsbracket=${value[1]}
number=$(extract_number_from_brackets "$nsbracket")
echo "${number}"
```

Listing B.5. `ima_ns_id_mapping.sh`

B.2 Counter solution 7.2

B.2.1 Templates and entries

Template for inserting the IMA namespace identifier and the counter value (Section 7.2.2).

```
static struct ima_template_desc builtin_templates[] = {
    {.name = IMA_TEMPLATE_IMA_NAME, .fmt = IMA_TEMPLATE_IMA_FMT},
    {.name = "ima-ng", .fmt = "d-ng|n-ng"},
    ...
    {.name = "ima-nsid-cnt", .fmt = "d-ng|n-ng|id|num_mes"},
};


static const struct ima_template_field supported_fields[] = {
    {.field_id = "d", .field_init = ima_eventdigest_init,
     .field_show = ima_show_template_digest},
    {.field_id = "n", .field_init = ima_eventname_init,
     .field_show = ima_show_template_string},
    {.field_id = "d-ng", .field_init = ima_eventdigest_ng_init,
     .field_show = ima_show_template_digest_ng},
    {.field_id = "d-ngv2", .field_init = ima_eventdigest_ngv2_init,
     .field_show = ima_show_template_digest_ngv2},
     ...
    {.field_id = "id",
     .field_init = ima_id_init,
     .field_show = ima_show_template_id},
    {.field_id = "num_mes",
```

```
     .field_init = ima_num_mes_init,
     .field_show = ima_show_template_id},
};
```

Listing B.6. `security/integrity/ima/ima_template.c`

Functions to handle the new entries.

```
int ima_id_init(struct ima_event_data *event_data, struct ima_field_data
    *field_data) {
    u32 ima_id = event_data->ima_ns_id;
    char string[MAX_LEN_ID];
    sprintf(string, "%u", ima_id);
    return ima_write_template_field_data(string, strlen(string),
        DATA_FMT_UINT, field_data);
}

int ima_num_mes_init(struct ima_event_data *event_data, struct ima_field_data
    *field_data) {
    u32 num_mes = event_data->num_measurements;
    char string[MAX_LEN_ID];
    sprintf(string, "%u", num_mes);
    return ima_write_template_field_data(string, strlen(string),
        DATA_FMT_UINT, field_data);
}
```

Listing B.7. `security/integrity/ima/ima_template_lib.c`

B.2.2 Atomic Parent extension

Implementation of a queue to standardize waiting times.

```
int ima_add_template_entry(struct ima_namespace *ns, struct
    ima_template_entry *entry, int violation, const char *op, struct inode
    *inode, const unsigned char *filename, int starting_ima_ns_id) {
    u8 *digest = entry->digests[ima_hash_algo_idx].digest;
    struct tpm_digest *digests_arg = entry->digests;
    const char *audit_cause = "hash_added";
    char tpm_audit_cause[AUDIT_CAUSE_LEN_MAX];
    int audit_info = 1;
    int result = 0, tpmresult = 0;
    if(starting_ima_ns_id == ns->id) {
        mutex_lock(&vett_queue_mutex);
        vett_queue[next_empty_slot] = ns->id;
        next_empty_slot = (next_empty_slot + 1)% MAX_VETT_QUEUE_LEN;
        mutex_unlock(&vett_queue_mutex);
    }
    while(vett_queue[actual_id] != starting_ima_ns_id);
    mutex_lock(&ima_extend_list_mutex);
    if (!violation && !IS_ENABLED(CONFIG_IMA_DISABLE_HTABLE)) {
        if (ima_lookup_digest_entry(ns, digest, entry->pcr)) {
            audit_cause = "hash_exists";
            result = -EEXIST;
            goto out;
        }
    }
    result = ima_add_digest_entry(ns, entry,
        !IS_ENABLED(CONFIG_IMA_DISABLE_HTABLE));
    if (result < 0) {
```

```
        audit_cause = "ENOMEM";
        audit_info = 0;
        goto out;
    }
    if(ns == &init_ima_ns)
        actual_id = (actual_id+1) % MAX_VETT_QUEUE_LEN;
    if (violation)
        digests_arg = digests;

    tpmresult = ima_pcr_extend(digests_arg, entry->pcr);
    if (tpmresult != 0) {
        snprintf(tpm_audit_cause, AUDIT_CAUSE_LEN_MAX, "TPM_error(%d)",
             tpmresult);
        audit_cause = tpm_audit_cause;
        audit_info = 0;
    }
out:
    mutex_unlock(&ima_extend_list_mutex);
    integrity_audit_msg(AUDIT_INTEGRITY_PCR, inode, filename,
               op, audit_cause, result, audit_info);
    return result;
}
```

Listing B.8. `security/integrity/ima/ima_queue.c`

Parent extension of the event (Section 7.2.2).

```
static int process_measurement(struct user_namespace *user_ns, struct file
    *file, const struct cred *cred, u32 secid, char *buf, loff_t size, int
    mask, enum ima_hooks func) {
    struct ima_namespace *ns;
    int ret = 0;
    u32 num_measurements = 0;
    int starting_ima_ns_id = 0;
    struct ima_namespace *starting_ima_ns;
    struct user_namespace *father = user_ns;
    if(user_ns != NULL) {
        starting_ima_ns = ima_ns_from_user_ns(user_ns);
        while(starting_ima_ns == NULL) {
            starting_ima_ns = ima_ns_from_user_ns(father->parent);
            father = father->parent;
        }
        starting_ima_ns_id = starting_ima_ns->id;
    }
    while (user_ns) {
        ns = ima_ns_from_user_ns(user_ns);
        if (ns_is_active(ns)) {
            int rc;
            num_measurements++;
            rc = __process_measurement(ns, file, cred, secid, buf, size, mask,
                func, num_measurements, starting_ima_ns_id);
            if(rc == 1) {
                break;
            }
            switch (rc) {
            case 0:
                break;
            case -EACCES:
                /* return this error at the end but continue */
```

```
                ret = -EACCES;
                break;
            default:
                /* should not happen */
                ret = -EACCES;
                WARN_ON_ONCE(true);
            }
        }
        user_ns = user_ns->parent;
    }
    return ret;
}
```

Listing B.9. `security/integrity/ima/ima_main.c`

B.3 Event solution 7.3

B.3.1 Templates and entries

Insertion of the new IMA templates and the related template entries (Section 7.3.2).

```
static struct ima_template_desc builtin_templates[] = {
    {.name = IMA_TEMPLATE_IMA_NAME, .fmt = IMA_TEMPLATE_IMA_FMT},
    {.name = "ima-ng", .fmt = "d-ng|n-ng"},
    {.name = "ima-sig", .fmt = "d-ng|n-ng|sig"},
    {.name = "ima-ngv2", .fmt = "d-ngv2|n-ng"},
    {.name = "ima-sigv2", .fmt = "d-ngv2|n-ng|sig"},
    ...
    {.name = "ns-event", .fmt = "imaeveninfo|imaidcreator|imaidcreated"},
};

static LIST_HEAD(defined_templates);
static DEFINE_SPINLOCK(template_list);
static int template_setup_done;
static const struct ima_template_field supported_fields[] = {
    {.field_id = "d", .field_init = ima_eventdigest_init,
     .field_show = ima_show_template_digest},
    {.field_id = "n", .field_init = ima_eventname_init,
     .field_show = ima_show_template_string},
    {.field_id = "d-ng", .field_init = ima_eventdigest_ng_init,
     .field_show = ima_show_template_digest_ng},
    {.field_id = "d-ngv2", .field_init = ima_eventdigest_ngv2_init,
     ...
    {.field_id = "imaidstart",
     .field_init = ima_eventinodexattrvalues_init,
     .field_show = ima_show_template_sig},
    {.field_id = "imaidmes",
     .field_init = ima_eventinodexattrvalues_init,
     .field_show = ima_show_template_sig},
    {.field_id = "imaidcreator",
     .field_init = ima_id_creator_init,
     .field_show = ima_show_template_string},
    {.field_id = "imaidcreated",
     .field_init = ima_id_created_init,
     .field_show = ima_show_template_string},
    {.field_id = "imaeveninfo",
     .field_init = ima_event_info_init,
```

```
    .field_show = ima_show_template_string},
};
```

Listing B.10. `security/integrity/ima/ima_template.c`

Functions that handle the creation of the custom entries.

```
int ima_id_creator_init(struct ima_event_data *event_data, struct
    ima_field_data *field_data) {
    u32 id_creator = event_data->ima_ns_id;
    char string[MAX_LEN_ID];
    sprintf(string, "%u", id_creator);
    return ima_write_template_field_data(string, strlen(string),
        DATA_FMT_UINT, field_data);
}

int ima_id_created_init(struct ima_event_data *event_data, struct
    ima_field_data *field_data) {
    u32 id_created = event_data->ima_ns_id2;
    char string[MAX_LEN_ID];
    sprintf(string, "%u", id_created);
    return ima_write_template_field_data(string, strlen(string),
        DATA_FMT_UINT, field_data);

}

int ima_event_info_init(struct ima_event_data *event_data, struct
    ima_field_data *field_data) {
    u32 event_info = event_data->event_info;
    char string[MAX_LEN_ID];
    sprintf(string, "%u", event_info);
    return ima_write_template_field_data(string, strlen(string),
        DATA_FMT_UINT, field_data);
}
```

Listing B.11. `security/integrity/ima/ima_template_lib.c`

B.3.2 Register the creation or closure of an IMA namespace

The function that creates the IMA entry that will be inserted in a parent (Section 7.3.2).

```
void ima_ns_event(struct ima_namespace *ns_creator, struct ima_namespace
    *ns_created, int event_info, struct ima_namespace *ns_to_extend) {
    int result = -ENOMEM;
    char *template_name = "ns-event";
    int template_num_fields = 3;
    int pcr = 10;
    char *template_fmt = "imaeveninfo|imaidcreator|imaidcreated";
    struct ima_template_entry *entry;
    struct ima_template_desc *template_desc = kzalloc(sizeof(struct
        ima_template_desc), GFP_NOFS);
    struct ima_event_data *event_data = kzalloc(sizeof(struct
        ima_event_data), GFP_NOFS);
    int violation = 0;
    const struct ima_template_field *info_ev =
        lookup_template_field("imaeveninfo");
    const struct ima_template_field *creator =
        lookup_template_field("imaidcreator");
```

```
    const struct ima_template_field *created =
        lookup_template_field("imaidcreated");
    const struct ima_template_field **fields_const = (const struct
        ima_template_field**)kzalloc(template_num_fields * sizeof(struct
        ima_template_field*), GFP_NOFS);
    fields_const[0] = info_ev;
    fields_const[1] = creator;
    fields_const[2] = created;
    event_data->event_info = event_info;
    event_data->ima_ns_id = ns_creator->id;
    event_data->ima_ns_id2 = ns_created->id;
    template_desc->fmt = kzalloc(sizeof(char)*(strlen(template_fmt)+1),
        GFP_NOFS);
    strncpy(template_desc->fmt, template_fmt, strlen(template_fmt));
    template_desc->name = kzalloc(sizeof(char)*(strlen(template_name)+1),
        GFP_NOFS);
    strncpy(template_desc->name, template_name, strlen(template_name));
    template_desc->num_fields = template_num_fields;
    template_desc->fields = fields_const;
    result = ima_alloc_init_template(event_data, &entry, template_desc);
    if (result < 0)
        return;
    result = ima_store_template(ns_to_extend, entry, violation, NULL, NULL,
        pcr);
    if (result < 0)
        ima_free_template_entry(entry);
    return;
}
```

Listing B.12. `security/integrity/ima/ima_api.c`

Parent extension of the event of IMA namespace creation.

```
static ssize_t ima_write_active(struct file *filp, const char __user *buf,
    size_t count, loff_t *ppos) {
    struct ima_namespace *ns = ima_ns_from_file(filp);
    struct user_namespace *user_ns_created = ima_user_ns_from_file(filp);
    struct ima_namespace *ima_ns_created =
        ima_ns_from_user_ns(user_ns_created);
    struct user_namespace *user_ns_creator = user_ns_created->parent;
    struct ima_namespace *ima_ns_creator =
        ima_ns_from_user_ns(user_ns_creator);
    struct ima_namespace *ima_ns_to_extend;
    struct user_namespace *to_extend = user_ns_created;
    unsigned int active;
    char *kbuf;
    int err;
        mutex_lock(&atomic_parent_extension_mutex);
    ...
    while(to_extend) {
        ima_ns_to_extend = ima_ns_from_user_ns(to_extend);
        if(ima_ns_to_extend != NULL && ns_is_active(ima_ns_to_extend))
            ima_ns_event(ima_ns_creator, ima_ns_created, 0, ima_ns_to_extend);
        to_extend = to_extend->parent;
    }
        mutex_unlock(&atomic_parent_extension_mutex);
    return count;
}
```

Listing B.13. `security/integrity/ima/ima_fs.c`

Parent extension of the event of IMA namespace closure (Section 7.3.2).

```
void free_ima_ns(struct user_namespace *user_ns) {
struct ima_namespace *ns = ima_ns_from_user_ns(user_ns);
struct user_namespace *parent_of_destroyed = user_ns->parent;
struct ima_namespace *ima_ns_to_destroy = ima_ns_from_user_ns(user_ns);
struct ima_namespace *ima_ns_parent_of_destroyed =
    ima_ns_from_user_ns(parent_of_destroyed);
struct user_namespace *to_extend = user_ns;
struct ima_namespace *ima_ns_to_extend;
    mutex_lock(&atomic_parent_extension_mutex);
ima_free_ima_ns(ns);
while(to_extend) {
    ima_ns_to_extend = ima_ns_from_user_ns(to_extend);
    if(ima_ns_to_extend != NULL && ns_is_active(ima_ns_to_extend))
        ima_ns_event(ima_ns_parent_of_destroyed, ima_ns_to_destroy, 1,
            ima_ns_to_extend);
    to_extend = to_extend->parent;
}
user_ns->ima_ns = NULL;
    ima_free_ima_ns(ima_ns_to_destroy);
    mutex_unlock(&atomic_parent_extension_mutex);
}
```

Listing B.14. `security/integrity/ima/ima_ns.c`

B.3.3 Measure process, atomic parent extension

Parent extension of a measure event.

```
static int process_measurement(struct user_namespace *user_ns, struct file
    *file, const struct cred *cred, u32 secid, char *buf, loff_t size, int
    mask, enum ima_hooks func) {
    struct ima_namespace *ns;
    int ret = 0;
    struct ima_namespace *starting_ima_ns = ima_ns_from_user_ns(user_ns);
    mutex_lock(&atomic_parent_extension_mutex);
    while (user_ns) {
        ns = ima_ns_from_user_ns(user_ns);
        if (ns_is_active(ns)) {
            int rc;
            rc = __process_measurement(ns, file, cred, secid, buf, size, mask,
                func, starting_ima_ns->id);
            switch (rc) {
            case 0:
                break;
        user_ns = user_ns->parent;
    }
    mutex_unlock(&atomic_parent_extension_mutex);
    return ret;
}
```

Listing B.15. `security/integrity/ima/ima_main.c`

B.4 nPCR solution 7.4

B.4.1 Templates and entries

For this solution, there is only one new template, used for the save of nPCR values (Section 7.4.2).

```
static struct ima_template_desc builtin_templates[] = {
    {.name = IMA_TEMPLATE_IMA_NAME, .fmt = IMA_TEMPLATE_IMA_FMT},
    {.name = "ima-ng", .fmt = "d-ng|n-ng"},
    ...
    {.name = "ima-dig-imaid", .fmt = "digev|imansid"}, /* placeholder for a
        custom format */
};

static LIST_HEAD(defined_templates);
static DEFINE_SPINLOCK(template_list);
static int template_setup_done;
static const struct ima_template_field supported_fields[] = {
    {.field_id = "d", .field_init = ima_eventdigest_init,
     .field_show = ima_show_template_digest},
    {.field_id = "n", .field_init = ima_eventname_init,
     .field_show = ima_show_template_string},
    {.field_id = "d-ng", .field_init = ima_eventdigest_ng_init,
     .field_show = ima_show_template_digest_ng},
    {.field_id = "d-ngv2", .field_init = ima_eventdigest_ngv2_init,
     .field_show = ima_show_template_digest_ngv2},
    {.field_id = "n-ng", .field_init = ima_eventname_ng_init,
     .field_show = ima_show_template_string},
     ...
    {.field_id = "digev",
     .field_init = digest_namespace_event_init,
     .field_show = ima_show_template_digest},
    {.field_id = "imansid",
     .field_init = ima_id_init,
     .field_show = ima_show_template_string},
};
```

Listing B.16. `security/integrity/ima/ima_template.c`

Functions to handle the new entries.

```
int digest_namespace_event_init(struct ima_event_data *event_data, struct
    ima_field_data *field_data) {
    u8 *cur_digest = NULL, hash_algo = ima_hash_algo;
    u32 cur_digestsize = 0;
    cur_digest = event_data->template_start_digest;
    cur_digestsize = VPCR_MAX_LEN;
    return ima_eventdigest_init_common(cur_digest, cur_digestsize,
        DIGEST_TYPE__LAST, hash_algo, field_data);
}

int ima_id_init(struct ima_event_data *event_data, struct ima_field_data
    *field_data) {
    u32 ns_ima_id = event_data->ima_ns_id;
    char string[MAX_LEN_ID];
    sprintf(string, "%u", ns_ima_id);
    return ima_write_template_field_data(string, strlen(string),
        DATA_FMT_UINT, field_data);
```

```
}
```

Listing B.17. `security/integrity/ima/ima_template_lib.c`

B.4.2 nPCR initialization

Initialization of all the nPCR for the namespace.

```
int ima_init_namespace(struct ima_namespace *ns) {
    int ret;
    int ret, i;
    ns->ns_status_tree = RB_ROOT;
    rwlock_init(&ns->ns_tree_lock);
    ns->id = ++counter_ns;
    for(i = 0; i < VPCR_MAX_LEN; i++)
        ns->vPCR[i] = 0;
    atomic_long_set(&ns->ima_htable.len, 0);
    atomic_long_set(&ns->ima_htable.violations, 0);
    memset(&ns->ima_htable.queue, 0, sizeof(ns->ima_htable.queue));
    INIT_LIST_HEAD(&ns->ima_measurements);
    ...
}
```

Listing B.18. `security/integrity/ima/ima_init_ima_ns.c`

B.4.3 nPCR extension

Performs the extension of the nPCR (Section 7.4.2).

```
int vprc_extension(u8 *vpcr_value, u8 *value_to_extend) {
    int rc;
    SHASH_DESC_ON_STACK(shash, ima_algo_array[ima_hash_algo_idx].tfm);
    shash->tfm = ima_algo_array[ima_hash_algo_idx].tfm;
    rc = crypto_shash_init(shash);
    rc = crypto_shash_update(shash, vpcr_value, VPCR_MAX_LEN);
    rc = crypto_shash_update(shash, value_to_extend, VPCR_MAX_LEN);
    rc = crypto_shash_final(shash, vpcr_value);
    return rc;
}
```

Listing B.19. `security/integrity/ima/ima_template_lib.c`

B.4.4 Measure of an event

```
int ima_add_template_entry(struct ima_namespace *ns, struct
    ima_template_entry *entry, int violation, const char *op, struct inode
    *inode, const unsigned char *filename) {
    u8 *digest = entry->digests[ima_hash_algo_idx].digest;
    struct tpm_digest *digests_arg = entry->digests;
    const char *audit_cause = "hash_added";
    char tpm_audit_cause[AUDIT_CAUSE_LEN_MAX];
    int audit_info = 1;
    int result = 0, tpmresult = 0;
    if(ns != &init_ima_ns) {
        mutex_lock(&(ns->npcr_lock));
        result = ima_add_digest_entry(ns, entry,
            !IS_ENABLED(CONFIG_IMA_DISABLE_HTABLE));
```

```
        tpmresult = vprc_extension(ns->vPCR, digest);
        host_extension_vpcr(ns->id, ns->vPCR);
        mutex_unlock(&(ns->npcr_lock));
        return result;
    }
    mutex_lock(&ima_extend_list_mutex);
    result = ima_add_digest_entry(ns, entry,
        !IS_ENABLED(CONFIG_IMA_DISABLE_HTABLE));
    ...
out:
    mutex_unlock(&ima_extend_list_mutex);
    integrity_audit_msg(AUDIT_INTEGRITY_PCR, inode, filename, op,
        audit_cause, result, audit_info);
    return result;
}
```

Listing B.20. `security/integrity/ima/ima_queue.c`

B.5 Keylime agent modifications

Function, written in `rust`, for the API that handles the container verification, the API is available at `/quotes/container`. The Agent implementation is the same for all the solutions (Section 8.2.2).

```
fn ima_ns_id_mapping(container_id: String) -> (String, String) {
    let script_file = "../../scripts/ima_ns_id_mapping.sh";
    let output =
        Command::new("bash").arg(script_file).arg(container_id).output().expect("failed
        to execute process");
    let out = output.stdout;
    let string = String::from_utf8(out).expect("Found invalid UTF-8");
    let mut line_vect: Vec<&str> = string.split("\n").collect();
    let ima_id = line_vect.remove(0);
    let mes_list_ns: String = line_vect.join("\n");
    if output.status.success() {
        (String::from(ima_id), String::from(mes_list_ns))
    } else {
        (String::from("error"), String::from("error"))
    }
}

pub async fn integrity_container(req: HttpRequest, param: web::Query<Integ>,
    data: web::Data<QuoteData>,
) -> impl Responder {
    ...
    if param.containerid.is_none() {
        warn!("Get quote returning 400 response. Not found the container
            identifier as parameter",
    );
        return HttpResponse::BadRequest().json(JsonWrapper::error(
            400,
            format!(
                "Not found the container identifier as parameter "
            ),
        ));
    }
    let mut container_identifier: String = param.containerid.clone().unwrap();
```

```
if !container_identifier.chars().all(char::is_alphanumeric) {
    warn!("Get quote returning 400 response. Parameters should be
        strictly alphanumeric: {}", param.mask);
    return HttpResponse::BadRequest().json(JsonWrapper::error(
        400,
        format!("mask should be strictly alphanumeric: {}", param.mask),
    ));
}
...
let tpm_quote = match context.quote(param.nonce.as_bytes(), mask,
    &data.pub_key, data.ak_handle, data.hash_alg, data.sign_alg,
) {
    Ok(tpm_quote) => tpm_quote,
    Err(e) => {
        debug!("Unable to retrieve quote: {:?}", e);
        return HttpResponse::InternalServerError().json(
            JsonWrapper::error(
                500,
                "Unable to retrieve quote".to_string(),
            ),
        );
    }
};
let id_quote = KeylimeQuote {
    quote: tpm_quote,
    hash_alg: data.hash_alg.to_string(),
    enc_alg: data.enc_alg.to_string(),
    sign_alg: data.sign_alg.to_string(),
    ..Default::default()
};
...
// Generate the measurement list
let (ima_measurement_list, ima_measurement_list_entry, num_entries) =
    if let Some(ima_file) = &data.ima_ml_file {
        let mut ima_ml = data.ima_ml.lock().unwrap(); //#[allow_ci]
        match ima_ml.read(
            &mut ima_file.lock().unwrap(), //#[allow_ci]
            nth_entry,
        ) {
            Ok(result) => {
                (Some(result.0), Some(result.1), Some(result.2))
            }
            Err(e) => {
                debug!("Unable to read measurement list: {:?}", e);
                return HttpResponse::InternalServerError().json(
                    JsonWrapper::error(
                        500,
                        "Unable to retrieve quote".to_string(),
                    ),
                );
            }
        }
    } else {
        (None, None, None)
    };
let mut ima_namespace_id: Option<String> = None;
let mut ima_measurement_list_namespace: Option<String> = None;
```

```
    let (ima_id, ima_ns_list) = ima_ns_id_mapping(container_identifier);
    if ima_ns_list == String::from("error") && ima_id ==
        String::from("error") {
        ima_namespace_id = Some((ima_id));
        ima_measurement_list_namespace = Some((ima_ns_list))
    }
    let quote = KeylimeQuote {
        pubkey,
        ima_measurement_list,
        mb_measurement_list,
        ima_measurement_list_entry,
        ima_namespace_id,
        ima_measurement_list_namespace,
        ..id_quote
    };
    let response = JsonWrapper::success(quote);
    info!("GET integrity quote returning 200 response");
    HttpResponse::Ok().json(response)
}
```

Listing B.21. `keylime-agent/src/quotes_handler.rs`

B.6 Keylime verifier and registrar modifications

The Keyliem verifier implementation varies from solution to solution because the verification logic changes.

B.6.1 Counter Solution verification

Class to handle the new IMA template.

```
class ImaNsIdCnt(Mode):
digest: Digest
path: Name
ns_id: int
counter: int
def __init__(self, data: str):
    tokens = data.split(" ", maxsplit=3)
    if len(tokens) != 4:
        raise ParserError(f"Cannot create imaNsIdCnt expected 2 tokens got:
            {len(tokens)}.")
    self.digest = Digest(tokens[0])
    self.path = Name(tokens[1])
    self.ns_id = int(tokens[2])
    self.counter = int(tokens[3])
def bytes(self) -> bytes:
    return self.digest.struct() + self.path.struct()
def is_data_valid(self, validator: Validator) -> Failure:
    return validator.get_validator(type(self))(self.digest, self.path,
        self.ns_id, self.counter)
```

Listing B.22. `keylime/ima/ima.py`

Validation of the list's entry (Section 8.3.2).

```
def _validate_ima_nsid_cnt(exclude_regex: Optional[Pattern[str]],
    runtime_policy: Optional[RuntimePolicyType], digest: ast.Digest, path:
    ast.Name, ns_id: int, cnt: int, hash_types: str = "digests",) -> Failure:
```

```
    failure = Failure(Component.IMA, ["validation", "ima-nsid-cnt"])
    if runtime_policy is not None:
        if exclude_regex is not None and exclude_regex.match(path.name):
            logger.debug("IMA: ignoring excluded path %s", path)
            return failure
        accept_list = runtime_policy[hash_types].get(path.name, None) # type:
            ignore
        if accept_list is None:
            logger.warning("File not found in allowlist: %s", path.name)
            failure.add_event("not_in_allowlist", f"File not found in
                allowlist: {path.name}", True)
            return failure
        hex_hash = digest.hash.hex()
        if hex_hash not in accept_list:
            logger.warning(
                "Hashes for file %s don't match %s not in %s",
                path.name,
                hex_hash,
                str(accept_list),
            )
            failure.add_event(
                "runtime_policy_hash",
                {
                    "message": "Hash not found in runtime policy",
                    "got": hex_hash,
                    "expected": accept_list,
                },
                True,
            )
            return failure
        if ns_id is None:
            failure.add_event("no ns_id present", f"does not have the
                identifier of the namespace", True)
            return failure
        if cnt is None:
            failure.add_event("no cnt present", f"does not have the counter of
                extensions", True)
            return failure
    return failure
```

Listing B.23. `keylime/ima/ima.py`

Verification logic.

```
import codecs
import hashlib
START_HASH = (codecs.decode('0'*40, 'hex'))
FF_HASH = (codecs.decode('f'*40, 'hex'))
START_HASH_256 = (codecs.decode('0'*64, 'hex'))
FF_HASH_256 = (codecs.decode('f'*64, 'hex'))
START_HASH_384 = (codecs.decode('0'*96, 'hex'))
FF_HASH_384 = (codecs.decode('f'*96, 'hex'))
START_HASH_512 = (codecs.decode('0'*128, 'hex'))
FF_HASH_512 = (codecs.decode('f'*128, 'hex'))
class Hash:
    SHA1 = 'sha1'
    SHA256 = 'sha256'
    SHA384 = 'sha384'
    SHA512 = 'sha512'
```

```
    supported_algorithms = (SHA1, SHA256, SHA384, SHA512)
    @staticmethod
    def is_recognized(algorithm):
        return algorithm in Hash.supported_algorithms
    @staticmethod
    def compute_hash(algorithm, tohash):
        return {
            Hash.SHA1: lambda h: hashlib.sha1(h).digest(),
            Hash.SHA256: lambda h: hashlib.sha256(h).digest(),
            Hash.SHA384: lambda h: hashlib.sha384(h).digest(),
            Hash.SHA512: lambda h: hashlib.sha512(h).digest(),
        }[algorithm](tohash)

start_hash = {
    Hash.SHA1: START_HASH,
    Hash.SHA256: START_HASH_256,
    Hash.SHA384: START_HASH_384,
    Hash.SHA512: START_HASH_512
}
ff_hash = {
    Hash.SHA1: FF_HASH,
    Hash.SHA256: FF_HASH_256,
    Hash.SHA384: FF_HASH_384,
    Hash.SHA512: FF_HASH_512
}
def extension_simulation(line, hash_alg):
    runninghash = start_hash[hash_alg]
    returning_list = []
    if line == '':
        return None
    tokens = line.split(" ")
    # check on the template
    if tokens[2] != "ima-nsid-cnt":
        return None
    number_of_extensions = int(tokens[-1])
    for x in range(1, number_of_extensions):
        value_4 = tokens[-4].split(':')
        byte_value = codecs.decode(value_4[1], 'hex')
        value_sha1 = value_4[0] + ':' +"\x00"
        value_to_hash = bytearray(value_sha1.encode('utf-8')) + byte_value

        value_3 = tokens[-3] + '\x00'
        value_to_hash = value_to_hash + bytearray(value_3.encode('utf-8'))

        value_to_hash = value_to_hash + bytearray((tokens[-2] +
            str(x)).encode('utf-8'))
        value_hashed = Hash.compute_hash(hash_alg, value_to_hash)
        runninghash = Hash.compute_hash(hash_alg, runninghash + value_hashed)
        final_string = tokens[0] + " " + codecs.encode(runninghash,
            'hex').decode('utf-8').lower() + " " + tokens[2] + " " + tokens[3]
            + " " + tokens[4] + " " + tokens[5] + " " + str(x)
        returning_list.append(final_string)
    return returning_list
```

Listing B.24. `keylime/ima/ima_extension.py`

B.6.2 Event Solution verification

Verification logic (Section 8.3.3).

```
import codecs
import hashlib
START_HASH = (codecs.decode('0'*40, 'hex'))
FF_HASH = (codecs.decode('f'*40, 'hex'))
START_HASH_256 = (codecs.decode('0'*64, 'hex'))
FF_HASH_256 = (codecs.decode('f'*64, 'hex'))
START_HASH_384 = (codecs.decode('0'*96, 'hex'))
FF_HASH_384 = (codecs.decode('f'*96, 'hex'))
START_HASH_512 = (codecs.decode('0'*128, 'hex'))
FF_HASH_512 = (codecs.decode('f'*128, 'hex'))
class Hash:
    SHA1 = 'sha1'
    SHA256 = 'sha256'
    SHA384 = 'sha384'
    SHA512 = 'sha512'
    supported_algorithms = (SHA1, SHA256, SHA384, SHA512)
    @staticmethod
    def is_recognized(algorithm):
        return algorithm in Hash.supported_algorithms
    @staticmethod
    def compute_hash(algorithm, tohash):
        return {
            Hash.SHA1: lambda h: hashlib.sha1(h).digest(),
            Hash.SHA256: lambda h: hashlib.sha256(h).digest(),
            Hash.SHA384: lambda h: hashlib.sha384(h).digest(),
            Hash.SHA512: lambda h: hashlib.sha512(h).digest(),
        }[algorithm](tohash)

start_hash = {
    Hash.SHA1: START_HASH,
    Hash.SHA256: START_HASH_256,
    Hash.SHA384: START_HASH_384,
    Hash.SHA512: START_HASH_512
}
ff_hash = {
    Hash.SHA1: FF_HASH,
    Hash.SHA256: FF_HASH_256,
    Hash.SHA384: FF_HASH_384,
    Hash.SHA512: FF_HASH_512
}
def extension_simulation(line, hash_alg):
for line in ima_entries_array:
    line = line.strip()
    if line == '':
        continue
    tokens = line.split()
    # initialization of the tree with the id of the host's ima namespace
    if tree == None and tokens[-1] != 0:
        tree = Node(tokens[-1])
    template_hash = codecs.decode(tokens[1], 'hex')
    if tokens[2] == 'ns-event':
        if str.isdigit(tokens[4]) and str.isdigit(tokens[5]):
            parent = int(tokens[4])
            child = int(tokens[5])
```

```
            else:
                print("wrong ima log structure")
                break

            if tokens[3] == 0:
                # creation of a namespace, insertion in the tree
                tree.add_child_between(tree.find_in_childs(parent),
                    tree.find_in_childs(child))
            elif tokens[3] == 1:
                # closure of the namespace
                tree.close_ns(child)
            else:
                print("wrong ima log structure")
                break
        if template_hash == start_hash[hash_alg]:
            template_hash = ff_hash[hash_alg]
        if tokens[2] == "ima-id":
            number_of_extensions = tree.node_heigh(tree.find_in_childs(tokens[5]))
            hash_mes = tokens[3].split(':')
            byte_value = codecs.decode(hash_mes[1], 'hex')
            value_sha1 = hash_mes[0] + ':' +"\x00"
            value_to_hash = bytearray(value_sha1.encode('utf-8')) + byte_value

            file_path = tokens[4] + '\x00'
            value_to_hash = value_to_hash + bytearray(file_path.encode('utf-8'))

            value_to_hash = value_to_hash + bytearray((tokens[-1]))
            value_hashed = Hash.compute_hash(hash_alg, value_to_hash)

            for x in range(0,number_of_extensions):
                runninghash = Hash.compute_hash(hash_alg, runninghash +
                    value_hashed)
        else:
            runninghash = Hash.compute_hash(hash_alg, runninghash + template_hash)
        found_pcr = (codecs.encode(runninghash, 'hex').decode('utf-8').lower() ==
            pcr_val.lower())
```

Listing B.25. `keylime/ima/ima_extension.py`

Custom library for IMA namespace creation tree

```
class Node:
    def __init__(self, id):
        self.childs = []
        self.id = id
        self.parent = None
        self.status = 'active'
    def add_child(self, child):
        self.childs.append(child)
        child.parent = self
        return child
    def add_child_between(self, parent, child):
        if(parent == None):
            return None
        parent.add_child(child)
        return
    def find_in_childs(self, value):
        found = None
        if self.id == value:
```

```
            return self
        for i in self.childs:
            if i.id == value:
                return i
            found = i.find_in_childs(value)
            if found != None:
                return found
        return found
    def find_in_tree(self, value):
        if(self.__find_in_childs(value) != None):
            return True
        else:
            return False
    def node_heigh(self, node):
        heigh = 0
        start = node
        while start != None:
            heigh += 1
            start = start.parent
        return heigh
    def close_ns(self, value):
        node_to_close = self.find_in_childs(value)
        if node_to_close == None:
            return -1
        node_to_close.status = 'close'
```

Listing B.26. `keylime/ima/tree_lib.py`

B.6.3 nPCR Solution verification

Class for verification of an entry regarding a nPCR value (Section 8.3.4).

```
class ImaDigNs(Mode):
    digest: str
    ima_ns_id: int
    def __init__(self, data: str):
        tokens = data.split(" ", maxsplit=1)
        if len(tokens) != 2:
            raise ParserError(f"Cannot create ImaDigNs expected 2 tokens got:
                {len(tokens)}.")
        self.digest = str(tokens[0])
        self.ima_ns_id = int(tokens[1])
    def bytes(self) -> bytes:
        return str.encode(self.bytes) + str.encode(str(self.ima_ns_id))
    def is_data_valid(self, validator: Validator) -> Failure:
        return validator.get_validator(type(self))(self.digest, self.path,
            self.ns_id, self.counter)
```

Listing B.27. `keylime/ima/ast.py`

The verification, for a namespace, is equal to a standard IMA measurements list verification and then compared with the nPCR value.

```
if "ima_namespace_id" in agent.keys() and "ima_measurement_list_namespace" in
    agent.keys():
    quote_vpcr = find_last_vPCR(agent["ima_measurement_list"],
        agent["ima_namespace_id"])
    ima_measurement_list_namespace = agent["ima_measurement_list_namespace"]
```

```
    quote_validation_failure =
        get_tpm_instance().check_quote(agentAttestState, agent["nonce"],
        received_public_key, quote_vpcr, agent["ak_tpm"],
        agent["tpm_policy"], ima_measurement_list_namespace, runtime_policy,
        algorithms.Hash(hash_alg), ima_keyrings, mb_measurement_list,
        agent["mb_refstate"], compressed=(agent["supported_version"] ==
        "1.0"))
    failure.merge(quote_validation_failure)
```

Listing B.28. `keylime/cloud_verifier_common.py`

Find the last nPCR value related to a given namespace in the host's list.

```
def find_last_vPCR(data: str, ns_id: int) -> str:
last_val: str = "";
data_arr = data.split("\n")
for element in data_arr:
    if element != "":
        tokens = element.split(" ")
        if int(tokens[4]) == ns_id:
            last_val = tokens[3]
return last_val
```

Listing B.29. `keylime/cloud_verifier_common.py`

B.7 Docker data

B.7.1 Dockerfile

```
FROM ubuntu:latest
COPY script_name.sh /
RUN chmod +x /script_name.sh
ENTRYPOINT ["/bin/bash", "/script_name.sh"]
```

Listing B.30. `Dockerfile`

B.7.2 Container script

```
exit_proc() {
    LSENT=$(ls -la /proc/1/ns | grep ima_ns)
    NSID=$(echo "$LSENT" | cut -d "[" -f 2 | cut -d "]" -f 1)
    cat /mnt/ima/ascii_runtime_measurements > where_save_list/$NSID
    exit 0
}
trap exit_proc SIGTERM SIGINT
mount -t securityfs none /mnt
echo 1 > /mnt/ima/active
echo -e 'measure func=BPRM_CHECK\nmeasure func=FILE_MMAP mask=MAY_EXEC\n'\
> /mnt/ima/policy
while true; do
sleep 10
done
```

Listing B.31. `script_name.sh`