POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Deep Attestation - Virtualise a Hardware-bound Trusted Platform Module

Supervisor
Prof. Antonio Lioy
Dr. Silvia Sisinni

Candidate
Alessandro De Crecchio

October 2023

A mia Mamma e mia Sorella

Summary

The increasing prevalence of digital infrastructures, particularly in the context of modern software-defined systems like cloud computing, has prompted a focus on optimizing hardware resource utilization and minimizing inefficiencies. However, this shift towards software-driven infrastructures brings forth security challenges, including software attacks like those exploiting side channels.

To counter these threats, technologies such as **Trusted Execution Environment** (TEE) and **Trusted Platform Module** (TPM) have been widely adopted. TEE offers a secure isolation environment for exceptionally dependable applications. Meanwhile, TPM, a secure and tamper-resistant coprocessor, boasts features like a random number generator and secure storage. These attributes enable the generation of an attestation proof essential for **Remote Attestation**, to verify the authenticity of both the hardware and software configurations, thereby ensuring the integrity of the currently operational applications.

The introduction of virtualization requires the virtualization of hardware modules like TPM to achieve the same level of security as non-virtualized systems. This necessitates research into binding virtual TPMs (vTPMs) instantiated for different virtual machines to the physical TPM (pTPM) at the hypervisor level. This procedure is known as **Deep Attestation**.

In literature, various solutions have been implemented to establish a strong connection between virtual TPM instances and physical TPM. The reasons for doing so are manifold, such as enhancing security against software vulnerabilities, which virtual TPMs inherently face due to their software nature, and establishing a robust and close association between virtual machines and their non-virtual counterpart, namely the hypervisor or VMM. Unfortunately, these solutions come with inherent tradeoffs involving vulnerabilities or infeasibilities.

This thesis aims to implement a Proof of Concept, proposed by Politecnico di Torino and HP, to establish a strong link between pTPM and vTPM for hardware-level protection during the Deep Attestation process. This concept should overcome limitations introducing a new TPM object, the Virtual Primary Seed (VPS), and features enabling hardware binding.

The work can be divided into different development phases corresponding to the different features implemented. After a study of the TPM Software Stack (TSS), a software system created to easily interact with TPM without having to worry about low-level details, and the internal methods of TPM, new methods were implemented and some existing ones were extended, both in the TSS and in the TPM code.

The implemented methods were then used to perform three basic operations for the solution: the creation of the vTPM instance with binding to the pTPM by creating a VPS for each hierarchy, the creation of objects internal to the vTPM that are both fully linked to the pTPM or fully softwarised to provide greater flexibility, and finally the shutdown and restart of the vTPM, which involves a storing and restoring operations of a permanent state.

The permanent state presented above includes all three procedures, as it contains the information about the binding (i.e. VPSs of the three key hierarchies), the different handles of the created objects and finally the NVRAM of the vTPM, which contains the handles to the permanently made objects and is therefore not to be lost when the vTPM is restarted.

These enhancements maintain compatibility with existing security workflows while allowing for the migration of sensitive vTPM areas to other pTPMs, thanks to duplicable vTPM primary keys.

Acknowledgements

I sincerely thank Prof. Antonio Lioy for giving me the opportunity to improve my knowledge by assigning this work, and I am sure it will be valuable for my near future.

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem statement

In these years, with the growth of the digital world, modern softwarized infrastructures have also increased in order to optimise and make full use of the hardware already available. This is the case, for example, with cloud computing: a new way of providing services in which the provider makes hardware resources available to developers and service providers. In this way, phenomena such as over-provisioning and waste of resources are reduced. Since this new paradigm relies heavily on virtualisation and thus on software implementations, and since there are many software attacks that can be carried out, the security and reliability of the infrastructure itself are among the most important requirements for a software infrastructure. Many countermeasures and methods have been developed to protect against software attacks. However, not all attacks are mitigated (e.g. attacks via side channels).

To defend against these types of attacks, technologies or devices that use hardware modules such as **Trusted Execution Environment** (TEE) and **Trusted Platform Module** (TPM) to implement remediation techniques are widely deployed. With TEE, it is possible to isolate an application that requires a high level of trustability in a protected environment. With TPM it is possible to implement defensive techniques such as **Remote Attestation** (RA): a method that allows a host to authenticate its hardware and software configuration to a trusted remote server. This technique is used to verify that an aplication running on another system behaves correctly by means of secure hardware that acts as a Root of Trust (RoT) (e.g. from TPM).

The TPM is a secure tamper-proof coprocessor and plays an important role in generating the attestation proof. It is not the trusted computing base (TCB) of a system but it allows to determine if TCB has been compromised. It implements an hardware random number generator, generations and storage of cryptographic keys to perform remote attestation and store data securely. Moreover it can be used also to authenticate hardware devices since each TPM chip has a unique and secret Endorsement Key. The attestation proof is generated starting from the state of the system. In order to store and report it the TPM leverage on its special registers named PCR: the core mechanism for recording platform integrity. PCRs are set

with a 0-value after each reboot or hardware signal and then they are updated as the system performs operations.

This context changes when it comes to virtualisation. As mentioned earlier, this is made possible by using a software layer such as the hypervisor to manage and organise hardware resources so that they can be allocated to virtual machines (VMs). Consequently, since TPM is a hardware module, it must also be virtualised in order to perform a Remote Attestation and achieve the same level of security as a non-virtualised system. Therefore, in order to protect the instance of the virtual TPM (vTPM) from the software attacks mentioned above, various researches have been carried out with the aim of linking the different vTPMs, instantiated for the different VMs, to the physical TPM (pTPM) at the hypervisor level. Various solutions have been proposed over the years but all of them seem to have some compromises. The procedure to perform VMs Remote Attestation is known as **Deep Attestation**.

Deep Attestation can be used to measure the Level of Assurance (LoA) of VMs and there are currently two solutions supported by ETSI [1]. The first is Single-channel VM-based Deep Attestation: in this model, the Remote Verifier establishes a communication channel with the VM and then requests attestation data from the VM and the hypervisor, the latter containing some specific vTPM data at the same time to address the binding between VM and vTPM. With a large number of VMs, this solution is no longer suitable as it takes a long time to generate all attestation reports. The second option is Multi-channel independent deep attestation: in this model, the remote verifier establishes a communication channel with the VM to retrieve only its attestation data, and then establishes another independent communication channel with the hypervisor to retrieve its attestation data. This solution provides a loose mapping between pTPM and vTPM, but is very suitable for the case where a large number of VMs are involved. The remote auditor will first certify all VMs and then perform the hypervisor attestation once.

1.2 Proposed solutions

This text work aims to implement a Proof of Concept of a solution proposed by Politecnico di Torino and HP [2] to establish a strong link between the pTPM and the vTPM to perform the Deep Attestation procedure with hardware level protection. So far, there are several proposals that pursue the same goal. Some of the most interesting are presented below and then analyzed later for the purposes of this work.

Berger et al. [3] in their work originally proposed a solution where a vTPM manager runs inside a privileged VM. This VM has hypervisor access to the pTPM and makes the already initialised vTPMs available to VMs that request them via drivers in the virtualised domain. To obtain hardware-level protection, the solution also provides for binding a vTPM identity credential to one of the pTPMs.

Wan et al. [4] discuss and investigate the limitations of TPM, which currently does not support multiple hosts or simultaneous access from multiple entities. This paper analyses a possible solution that would enable paravirtualisation of TPM. In

this way, the pTPM would not need any further implementations to be accessible to VMs in a virtualised environment.

The solution proposed by Orange [5] implements Deep Attestation by means of cryptography. It can be placed halfway between single and multiple channel, its scheme aims to attest hypervisor and VMs only once. Within the hypervisor attestation nonce is embedded a list of public keys associated to VMs, managed by hypervisor, generated by the RoT. The goal of this solution is to individually attest components and have these attestation proof linked to hypervisor. This solves the problem of multiple VMs attestation that run in a single hypervisor. However a weak point of this is that actually it limits some features provided by TPM such as: migrating VMs to another hypervisor or replacing TPMs.

More recently, new tools and technologies have been developed to address this problem. The tool Keylime provides vTPMs in software modules that exist on the host machine and are exported as devices to QEMU/KVM-based VMs. Wang et al. [6] in turn have proposed the use of Intel's TEE to protect vTPMs within SGX enclaves, i.e. a protected memory area where only trusted code is allowed. The latter as cons, like any software component, can be subject to attacks such as side channels.

This thesis work aims to implement a Proof of Concept that should overcome the problems presented so far. In this work the design implementation of a TPM extension and key features that enable the hardware-binding are discussed. TPM extension should overcome limitations listed before by adding a new TPM object to enable hardware protection of vTPM primary seeds, wrapping vTPM primary keys and create child object, both hardware bound both fully softwarised. These enhancements do not change the vTPM life-cycle and commands, so existing security workflows do not need to change or upgrade their behaviour. In addition all sensitive areas of vTPM can be migrated to another pTPM, since the vTPM primary keys in this solution are duplicable.

1.3 Thesis structure

In this work provides the necessary information to understand the solution and the scenario in which it is applied. A particular focus will be on virtualised environments in conjunction with trusted computing. We will analyse the TPM and the related attestation procedures, then we will examine the latter in a virtualised scenario, highlighting the differences. Subsequently, the extensions for the realisation of the PoC and their implementations will be presented.

The next chapters of the Thesis can be summarised as follows:

- **Chapter 2:** In this chapter background information on Trusted Computing technologies and brief presentation of the current state of the art are given. The focus is on TPM and its internal structures and architectures by presenting and describing one of the procedures in which it plays the main role: Remote Attestation.

- **Chapter 3:** In this chapter it is discussed about security in virtualised environments, i.e. those of cloud computing. It gives an introduction to the cloud computing paradigm and the associated security requirements. Methods are also presented with which a virtualised environment can be made secure and trusted through the use of new solutions.

- **Chapter 4:** In this chapter it is analyzed Deep Attestion, in particular the problem of VMs and hypervisor attestation. It is presented the entire processes and architecture to perform this kind of attestation. Different solutions to these problems are also studied, exploring their strengths and weaknesses.

- **Chapter 5:** In this chapter it is explained the solution to implement, detailed description of the TPM extension to add in order to gain the necessary features is provided. In particular it is presented how multiple vTPM are mapped to a single pTPM by adding their primary seed object within the pTPM hierarchies. It is also described how to generate fully softwarised object, i.e. without any binding to pTPM, and how migrate a VM to another host equipped with another pTPM.

- **Chapter 6:** In this chapter standard pTPM commands to be extended are described. Then the extension commands are presented: these are fundamental to initialise and use vTPM and to obtain the features just described. The initialisation and key creation procedures, with all the steps required for each vTPM instantiation, as well as saving and restoring the state when a hardware connection is required, are explained.

- **Chapter 7:** In the last chapter is possible to find conclusions about the work and some possible future works are presented to add other features to the actual implementation.

Chapter 2

Trusted Computing and Remote Attestation

In the dynamic landscape of information technology, where industries are experiencing unprecedented growth, the prevalence of security incidents within information systems has become an alarming reality. These incidents carry far-reaching implications, not only for individual users but also for national security and societal stability. Safeguarding our national information security has thus become an imperative.

To address this escalating challenge, a crucial requirement arises: the need for users to assess the credibility of the entities they interact with in cyberspace. Ensuring secure interactions in this digital realm hinges upon trust as its foundational pillar.

Given that trust forms the bedrock of secure interactions in cyberspace, it follows that all facets of information processing and interaction within this context must be inherently trusted. Consequently, the establishment of measures to guarantee the credibility of cyberspace itself is of paramount importance, serving as the cornerstone of the concept known as Trusted Computing.

2.1 Trusted Computing concepts

Trusted computing, a paradigm pioneered and standardized by the Trusted Computing Group, strives to instill unwavering trustworthiness in computing platforms. It accomplishes this ambitious goal by constructing a comprehensive chain of trust, a meticulous record which includes all the hardware and software components utilized in a given context. Unlike conventional methods, such as virus scanners that seek to detect and eliminate malicious software, this approach focuses on contrasting the established chain of software with a list of trusted applications.

In 1999, the Trusted Computing Platform Alliance (TCPA) emerged as a consortium comprising various technology giants committed to advancing trust and security in computing platforms. This alliance boasted some of the tech industry's heavyweights, including Hewlett-Packard, Microsoft, Intel, and IBM, and it swiftly

garnered over 70 members within its inaugural month of operation. Under the TCPA's aegis, a set of Trusted Computing Platform Specifications was conceived. These specifications served as a guiding beacon for the industry, facilitating the establishment of trust within computing platforms and ecosystems. They mandated the implementation of specific mechanisms and processes, thus ensuring the security of computing systems. In the wake of the TCPA, the Trusted Computing Group (TCG) emerged in 2003, with a renewed focus on bolstering mobile security. The TCG continues to operate to this day, steadfast in its mission to develop and advocate for specifications aimed at shielding computer resources from threats posed by malicious entities, all while respecting the rights of end users.

Nowadays, the principles that TCG believes underlie the effective, useful, and acceptable design, implementation, and use of TCG technologies are the following:

1. Security: TCG-enabled components should achieve controlled access to designated critical secured data and should reliably measure and report the system's security properties. The reporting mechanism should be fully under the owner's control.

2. Privacy: TCG-enabled components should be designed and implemented with privacy in mind and adhere to the letter and spirit of all relevant guidelines, laws, and regulations.

3. Interoperability: Implementations and deployments of TCG specifications should facilitate interoperability. Furthermore, implementations and deployments of TCG specifications should not introduce any new interoperability obstacles that are not for the purpose of security.

4. Portability of data: Deployment should support established principles and practices of data ownership.

5. Controllability: Each owner should have effective choice and control over the use and operation of the TCG-enabled capabilities that belong to them; their participation must be opt-in. Subsequently, any user should be able to reliably disable the TCG functionality in a way that does not violate the owne's policy.

6. Ease-of-use: The nontechnical user should find the TCG-enabled capabilities comprehensible and usable.

The core technologies proposed by the Trusted Computing Group revolve around the Trusted Platform Module (TPM). These technologies are designed to usher in a new era of trust in computing, assuring users and organizations alike of secure and reliable digital interactions. Before we delve into the TPM and its functions, it's essential to introduce some fundamental concepts within this context.

Trusted Computing Base

Abbreviated as TCB, it represents a collective set of system resources, encompassing both hardware and software, responsible for upholding the system's security

policies. Crucially, the TCB possesses the ability to protect itself from compromise by any external hardware or software components that do not form part of the TCB. The TCB shields itself from external influences, including both self-tests and external hardware Roots of Trust, rendering them immune to modification by any hardware or software elements, save for physical tampering.

Root of Trust

This critical element serves as the foundation for establishing trust within a system. It is characterized by its unwavering, expected behavior, which, if compromised, remains undetectable. Typically, a Root of Trust comprises a hardware component, and in some instances, it may also involve software elements. Within the domain of trusted computing, several Root of Trusts exist:

- Root of Trust for Measurement (RTM): The RTM is responsible for measuring and calculating values that determine the system's integrity. These measurements are then securely conveyed to another Root of Trust, often referred to as the Core Root of Trust for Measurement (CRTM), which is typically a software component executed by the CPU.
- Root of Trust for Storage (RTS): This is a specialized section of memory that enjoys a heightened level of protection. It's shielded to prevent any entity, except the CRTM, from altering its contents.
- Root of Trust for Reporting (RTR): The RTR is tasked with securely reporting the contents of the RTS. In essence, the RTM calculates the measurement, securely stores it in the RTS, and when needed, the RTR retrieves and shares this measurement with an external verifier.

It is important to distinguish two different types of Root of Trust Measurement: Static RTM (S-RTM) and Dynamic RTM (D-RTM). Both are defined by the TCG as ways to establish trust during boot.

As implied by its name, the concept of Static Root of Trust for Measurement hinges on commencing trust from a fixed, unalterable piece of code known as the Core Root of Trust for Measurement (CRTM). In typical computing platforms, the initial component that springs to life is the BIOS. Therefore, for the establishment of trust within a computing platform, an additional entity is necessitated to measure the BIOS and undertake the role of a CRTM. This entity serves as a foundational Trusted Building Block (TBB) that retains its unchanging nature throughout the platform's entire lifecycle. The CRTM can take different forms: it might be seamlessly integrated into the BIOS itself, resembling a BIOS boot block. Alternatively, it can manifest as a set of CPU instructions, typically residing within a chip on the motherboard. However, it's essential to recognize that when using the S-RTM, we are primarily checking the integrity of the machine at load-time. This signifies that we can ascertain and place trust in what is loaded onto the system but not necessarily in what is executed thereafter.

Dynamic Root of Trust for Measurement (D-RTM) introduces a significant refinement in the TCB, simplifying the assessment of the platform's state, thus enhancing manageability. Unlike its static counterpart, D-RTM adopts a more flexible approach to establishing trust in computing components. Talking about D-RTM the trustworthiness of various system elements is temporarily set aside until a secure event transpires. For instance, the launch of a hypervisor in a secure manner serves as one such event, initiating the process and initializing the system. At this juncture, the initial root of trust measurement is initiated, marking the inception of trust assessment.

What distinguishes D-RTM from S-RTM is its ability to exclude components that were previously staged before the secure event. These pre-staged components are deliberately omitted from the TCB, preventing their execution after the trust properties of the system have been established. In essence, D-RTM streamlines the trust establishment process, making it more adaptable and efficient when compared to the static counterpart, S-RTM.

Chain of Trust

As the name suggests, this concept entails a sequential chain of components that engender trust and facilitate a comprehensive system audit. Typically, Component A measures Component B, and the resultant measurements are stored in the RTS. If Component A is deemed reliable, the verifier can discern the integrity of Component B because the anticipated hash value is already known.

2.2 TPM 2.0 structures and operations

The concept of TPM emerged in the late 1990s. The TCG developed the TPM specification to address new security challenges. TPM 1.0 was introduced in 2000, followed by subsequent versions, including TPM 1.2 and TPM 2.0, which brought significant enhancements in terms of functionality and security. It is a tamper-resistant chip, although not tamper-proof, certified at the Common Criteria EAL4+ level and equipped with various cryptographic modules. TPM technology has evolved over the years, becoming an essential component in modern cybersecurity strategies.

2.2.1 TPM implementations and features

TPM-2.0 can be implemented in various way:

- Discrete: implements functionality in a semiconductor package as dedicated chip, among manufacturers we can find STM or Infineon.
- Integrated: the hardware components of the TPM is embedded as part of another chip, as in the case of Intel.
- Firmware: the firmware itself run in a Trusted Execution Environment. AMD, Intel and Qualcomm have implemented firmware TPM.

- Hypervisor: runs in an isolated execution environment, as in the case of firmware TPM.
- Software: emulator running in user space, quite useful for development purposes, it is not secure.

There are several features of this chip, some are listed below.

TPMs incorporate a true hardware random number generator, as opposed to a pseudo RNG. They come equipped with cryptographic capabilities, enabling essential operations like key generation, encryption, and digital signatures. These operations take place within the secure enclave of the TPM, rendering them resilient to external attacks.

Furthermore, TPMs offer the capacity to securely store sensitive data, including encryption keys and digital certificates. These keys remain shielded against unauthorized access, even if the host system faces a compromise. TPMs also introduce the concept of data sealing, permitting the encryption of data to a specific system configuration. Unsealing this data requires a matching system configuration, providing an influential mechanism for data protection.

TPMs extend support for Remote Attestation, a feature allowing remote entities to verify the integrity of a device's software and firmware. This capability proves invaluable in scenarios like cloud computing and the Internet of Things (IoT).

In terms of hardware authentication, computers can leverage TPMs to authenticate hardware devices since each TPM chip possesses a unique and confidential Endorsement Key (EK), which is established during production. While this uniqueness ensures precise machine identification, it simultaneously raises privacy concerns. It becomes imperative to ensure that this feature is exclusively accessed by authorized individuals.

2.2.2 Key hierarchies, objects and registers

When describing TPMs, a key aspect of its architecture revolves around key hierarchies, objects, and PCRs (platform configuration registers). These elements form the foundation upon which the TPM's security and encryption operations are based.

Key hierarchies

Rather than having fixed keys, TPM uses three distinct key hierarchies, each potentially comprising multiple keys and a variety of algorithms. Each hierarchy boasts dedicated authorization mechanisms, including, at a minimum, a password, and it is governed by specific policies that can range from simple to complex. Furthermore, each hierarchy operates with a distinct seed for generating primary keys, ensuring that keys within different hierarchies remain independent of each other.

These hierarchies include the following:

1. Platform Hierarchy: The platform hierarchy is associated with the board or system that hosts the TPM, using it as both the Root of Trust for Storage (RTS) and the Root of Trust for Reporting (RTR). Its primary purpose is to manage firmware-related components, housing non-volatile storage, keys, and data pertinent to the platform's operation.

2. Endorsement Hierarchy: The endorsement hierarchy serves as a dedicated repository for keys and data related to privacy administration. Privacy administrators utilize this hierarchy to safeguard sensitive information.

3. Storage Hierarchy: Typically managed by the platform's owner, who often also serves as the privacy administrator, the storage hierarchy comprises non-volatile storage, keys, and data. It plays a pivotal role in securely storing and managing critical components within the TPM.

Objects

An object has three parts [7]: the public, private area which reside inside TPM itself and sensitive one that is external. Sensitive area may not be present, while the other two are mandatory.

Public Area: It serves the purpose of uniquely identifying an object, and even without any specific permissions, it allows for the listing of stored objects within it.

Private Area: Housed exclusively within the TPM, this area stores the secrets associated with the object.

Sensitive Area: An encrypted private area, this area facilitates storage sensitive data external to the TPM while maintaining data security. The sensitive area enables the configuration of an authorization value, such as a password, for the newly created object.

Objects are characterized by some information that can be specified using a template [7]. This template contains the following:

- The object type, which can indicate whether it's a symmetric key, HMAC key, asymmetric key, or a data value.
- The name's algorithm, which identifies the hash algorithm used to compute the object's Name).
- The object attributes, which consist of a set of flags that determine how the object can be used and the rules governing its loading into the TPM.
- An authorization policy required to access the object.
- A unique value that may be employed by the TPM during the object creation process, serving as a user-provided argument.

The object's attribute flags [7] can be in either a set or clear state and collectively form a bit-mask. These attributes are the following:

- fixedTPM: When set, it prohibits the object from being duplicated.
- fixedParent: If clear, the key itself can be duplicated. When set and combined with a clear fixedTPM, the object may be migratable, contingent on one of its parent objects being duplicable.
- restricted: When set, the object (which is inherently a key) is limited to performing cryptographic operations solely on data structures with known formats. For instance, a restricted signing key cannot sign arbitrary values; it can only process data that has been previously hashed by the TPM itself.
- sensitiveDataOrigin: If set, the TPM generates the sensitive data (e.g., a symmetric key). Otherwise, the sensitive data must be provided by the caller.
- sign: When set, the key can be utilized for signing. In the case of a symmetric algorithm, this entails conducting an HMAC computation. For symmetric keys, this flag also governs the ability to perform encryption.
- decrypt: When set, the key can be employed for decryption. Encryption is carried out using the public key for asymmetric algorithms, and for symmetric algorithms, it employs the same key.

The objects managed by the TPM are basically three: Primary Keys, Sealed Data Objects (SDO) and Ordinary Keys.

Primary keys, namely endorsement keys and storage keys, are generated from one of the primary seeds. The TPM never discloses the private value, ensuring that the private keys remain securely stored within the TPM, preventing any external extraction. In the event that the private key is lost or destroyed, it can be regenerated using the same seed and parameters, provided that the primary seed remains unaltered. They are generated inside the TPM in a deterministic manner when they are required. This process involves utilizing the primary seed and a template as input.

Sealed Data Objects are essentially data encrypted based on certain conditions. This is accomplished through a combination of the keyed hash object type and both the sign and decrypt attribute clear, so the object is not intended to sign or encrypt other data. Additionally, the restricted flag must also be clear, as the SDO is usually generated from a data blob provided by the user as part of the sensitive area, alongside the template and the parent key's handle. Since TPM 2.0 offers context management an SDO can be swapped out of context as needed, then it is securely stored encrypted with a symmetric key. However, a significant limitation of storing sensitive information within an SDO is its size constraint. According to specifications, a keyed hash data type must has maximum limit of 128 bytes.

Ordinary Keys and SDOs are safeguarded through the use of a Storage Parent Key (SPK). Within the TPM, the presence of this SPK is indispensable for the loading or creation of any key or SDO. The generation of randomness required for these keys is sourced from the TPM's internal Random Number Generator (RNG). However, it's crucial to note that when the TPM returns these keys' private components, they are protected by the SPK. As a result, the private portion must

be stored externally for safekeeping. Yet, there's a critical dependency on the TPM, the SPK is imperative for decrypting this private information when needed.

Registers - PCR

The TPM serves the function of reporting the current system state, and for both storing and conveying this state, it incorporates a distinct set of registers known as PCR (Platform Configuration Registers) [7]. These registers serve as repositories for maintaining the historical record of the platform's configuration, essentially functioning as the core mechanism for recording platform integrity. It's vital to note that these registers experience a reset solely during a platform reset event, such as a reboot or a hardware signal, causing all the registers to revert to their initial state of 0. This is a significant security measure, as it prevents malicious code from manipulating these registers to revert them to certain values.

The operations performed on these registers are limited to two: reset and extend. The extend operation involves a process where the old value contained within the PCR is concatenated with new data and then hashed, resulting in the generation of a new value for the PCR. According to the Trusted Computing Group for a PC client the PCRs are used for various purposes.

2.2.3 TPM2 Software Stack

The TSS [8] is a software system built to make it easier for TPM application programmers to work with the TPM without needing to worry about low level details. The TSS has different layers, which makes it flexible. Figure 2.1 shows all the layers of the TSS, which are described below.

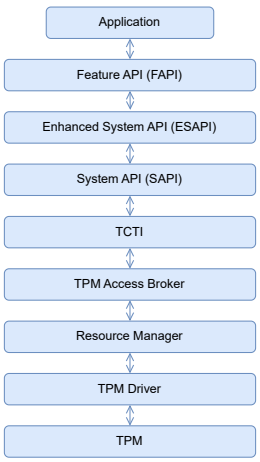

Figure 2.1. TPM Software Stack

Feature API (FAPI)

The Feature API [9] offers a higher-level software abstraction for application developers, shielding them from low level implementations. For example, an application

may need to create a key without delving into the technical details. This level of abstraction is provided by the feature and application APIs, making it easier for developers to work with TPM functionality without requiring extensive knowledge of TPM internals.

Enhanced System API (ESAPI)

The Enhanced System API [10] primary purpose is to simplify the complexity for applications that require individual system level TPM calls while also necessitating cryptographic operations on the data exchanged with the TPM. ESAPI is particularly useful for applications utilizing secure sessions to perform tasks like Hash-based Message Authentication Code (HMAC) operations, parameter encryption, decryption, TPM command auditing, and TPM policy operations. Additionally, ESAPI provides context and object management.

System API (SAPI)

The System API [11] serves as a foundational layer in the TSS architecture, granting access to the full range of functionality offered by a TPM 2.0 implementation. It is designed for use in scenarios where low-level calls to TPM functions are necessary, spanning firmware, BIOS, applications, operating systems, and more. The System API, being a low level interface, is primarily intended for expert usage.

TPM Command Transmission Interface (TCTI)

The TPM command transmission interface [12] handles all communication to and from the lower layers of the stack. It provides different interfaces for various TPM types, including local hardware TPMs, firmware TPMs, virtual TPMs, remote TPMs, and the TPM simulator. Additionally, it offers two distinct interfaces to TPMs: the legacy TIS interface and the command/response buffer (CRB).

TAB and Resource Manager

The resource manager operates in a way similar to a virtual memory manager, overseeing TPM context management. It effectively swaps objects, sessions, and sequences in and out of the TPM's limited onboard memory as needed. While this layer remains transparent to the upper layers of the TSS, it is not obligatory. If not implemented, the upper layers will assume responsibility for managing TPM context. The TPM access broker [13] manages multi-process synchronization for TPM access, ensuring that a process accessing the TPM can complete its command without interference from other concurrent processes.

TPM Device Driver

The TPM device driver serves as the operating system specific driver responsible for managing the interaction with the TPM. It handles tasks such as communication with the TPM and the reading and writing of data to it.

2.3 Measurements and Remote Attestation

Remote attestation, a crucial element in the pursuit of trust, relies on accurately measuring a system's components and their configurations. This process offers a means for remote entities to evaluate the integrity and security of an information platform without relying solely on self-declarations.

At the core of remote attestation lies system measurement, a fundamental practice in which the specific attributes of a computer system are assessed, quantified, and encapsulated in an auditable report. These measurements encompass critical elements such as firmware, bootloader, operating system, and even application code, providing a comprehensive overview of the system's security status.

In this section, we will delve deeply into the examination of Measured Boot and Linux IMA [14], both types of measurement for a system. Subsequently, we will dissect the flow of Remote Attestation.

2.3.1 Measured Boot

Measured Boot fundamentally serves as a security mechanism that guarantees the verifiability of a system's boot process. This is achieved by generating cryptographic measurements, or "hashes," at various stages of the boot sequence, effectively creating a digital fingerprint of the system's state. These measurements capture essential information about the software and firmware components involved in the boot-up process, including the BIOS, bootloader, and the operating system kernel. The key to Measured Boot lies in securely storing these measurements within a hardware-based security module (HSM). Consequently, once a system completes the boot process, the entire history of boot-time events can be cryptographically verified, thereby ensuring that no unauthorized or malicious changes have occurred. Measured Boot plays a fundamental role in system attestation, allowing remote parties to confirm the authenticity and integrity of a computing device. It effectively prevents the infiltration of rootkits, malware, and other forms of compromise during system startup.

To implement this mechanism, using a TPM as HSM, different steps are required as shown in Figure 2.2:

1. The first stage of boot loader, assumed to be trustworthy, forms the CRTM. This values is stored in the PCR.

2. The boot loader will calculate the hash of the second stage boot loader (UEFI/BIOS) and subsequently load and execute it. The corresponding value is then recorded in the PCR through an Extend Operation by combining the hash of first and second measure into $h(M1||M2)$.

3. The second stage boot loader measures and loads the operating system, resulting in $h(M1||M2||M3)$ stored in the PCR. The PCR functions as an accumulator, not merely summing values, but instead, accumulating hashes recursively. The final PCR values depend on all the inserted hash values and their sequence.

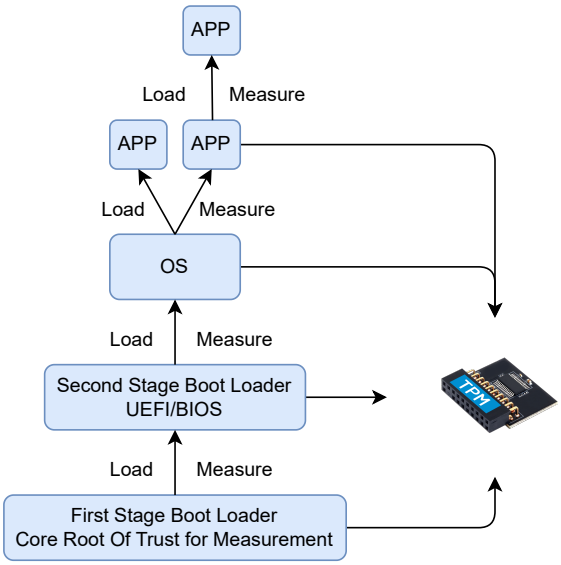

Figure 2.2. Measured Boot

4. The operating system can be configured to follow a similar procedure. Whenever an application is launched, it undergoes measurement, loading, and extension into the PCR. This process is optional and varies across operating systems.

Through this implementation within the PCR, a comprehensive history of executed processes on the system, starting from the boot, is preserved. This history defines the system's state, representing the sequence of applications executed. Consequently, applications can load, measure, and extend other applications, perpetuating the PCR's record of the system's state.

2.3.2 Linux IMA

IMA, or the Integrity Measurement Architecture, is an integral component embedded within the Linux operating system, tasked with executing a range of operations. IMA operates by establishing a comprehensive framework for measuring and verifying the integrity of critical system files, applications, and configurations. This technology generates cryptographic hashes, or measurements, for each file and component as they are accessed or executed within the Linux system. These measurements serve as digital fingerprints, enabling real-time validation of the authenticity and integrity of the software components.

Linux IMA is not just about passive measurements; it empowers administrators with the ability to define and enforce security policies that determine which files and applications are allowed to run based on their trustworthiness. It provides the means to enforce access controls and implement security policies that can protect the system from unauthorized changes or tampering.

In detail, the operations it performs are the following:

- Collect: Whenever the Linux kernel is invoked for an "exec" operation, IMA performs a measurement of the binary before it's executed.
- Store: Following the measurement, it stores this data within a designated Platform Configuration Register (PCR), specifically within PCRs 8 to 15. This entails adding the measurement to a list resident within the kernel and extending the IMA PCR.
- Appraise (not mandatory): It enforces local validation by comparing the measurement to an expected "good" value stored in an extended attribute of the file. Certain Linux file systems allow the addition of extended attributes to files, in addition to their standard attributes such as modification time, owner, and group. When this feature is active, the expected "good" value can be stored. Consequently, when the binary is executed, its hash should match the expected value; if not, execution is denied. It's worth noting that this appraisal step is optional, but the integrity of these unmodified data must be trusted.
- Protect: IMA ensures the security of a file's extended attributes, including the appraisal hash, against offline attacks. In the event someone attempts to modify these extended attributes through an application that allows such changes, they may succeed. However, when a new application is executed, any discrepancies will be detected.

Linux IMA uses PCR10 as its storage destination, and it kickstarts PCR10 with the initial measurement of the "boot_aggregate." This signifies that when it begins its operation, Linux IMA calculates the hash of all TPM's PCR values, ranging from 0 to 7, which report on the correctness of the boot process. The specific elements that IMA is expected to measure are outlined by the IMA template, often utilizing the "ima-ng," though customization is also possible. The kernel's securityfs maintains a comprehensive list of all measurements carried out by IMA.

2.3.3 Remote Attestation flow

Remote attestation (RA) represents a security service that empowers a remote Verifier to judge the condition of an untrusted remote Prover, often a device. The spectrum of remote attestation paradigms extends from purely software-based approaches to exclusively hardware-centric solutions. Hybrid attestation models find their place in between, harnessing the enhanced security capabilities of secure hardware components while benefiting from the cost-effectiveness of software-based implementations. Traditional remote attestation protocols primarily focus on assessing the state of a Prover device.

One of our key areas of focus centers around hardware-based Remote Attestation. Hardware-based remote attestation leverages physical chips and dedicated modules to establish trust in remote systems. These specialized modules include components like TPMs, which are hardware modules designed to facilitate secure storage and computations.

Remote attestation bases its protocol on some key properties.

Fresh information: An attestation should attest the current state of the Prover.

Comprehensive information: Attestation mechanisms should possess the capability to provide data that enables the Verifier to judge the condition of the Prover.

Trustworthy mechanism: The Verifier should possess the capability to ascertain the accuracy of information provided by the Prover, even when faced with an actively hostile adversary.

Exclusive access: The attestation mechanism of the Prover should be the sole entity with read access to the secret "k". No other process or device should be granted permission to access or retrieve the value of secret "k".

No leaks: The attestation mechanism must prevent any disclosure of information that would enable an adversary to infer the secret "k".

Immutability: The attestation mechanism remains unalterable, impervious to any tampering attempts, whether initiated by a local or remote adversary with access to the device.

Atomic execution: The attestation mechanism's execution remains uninterrupted, impervious to any actions initiated on the device.

Controlled invocation: The attestation mechanism should exclusively be activated from its designated entry point.

The Remote Attestation protocol requires both a Verifier and a Prover, and a prevalent approach to implementing RA is through a challenge-response protocol. A simple scheme of that protocol is shown in Figure 2.3.

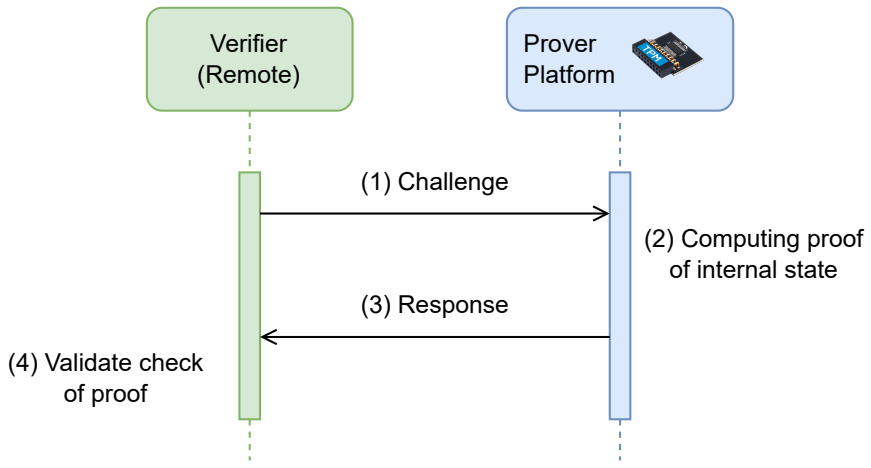

Figure 2.3. Remote Attestation protocol

This interaction can be structured as a 3-step process that the Prover and Verifier jointly undertake, encapsulated in the operations (Request, Attest, Verify).

- Request(): The Verifier initiates the process by generating a random challenge, referred to as "Chal," which is then transmitted to the Prover.

- Attest(Chal): Subsequently, the Prover engages in self-attestation and incorporates the challenge into the digest. The Prover then forwards this digest to the Verifier.
- Verify(Digest, Chal): The Verifier, in the final step, assesses the validity of the response provided by the Prover. Knowledge of the Prover's legitimate internal states is essential in this phase to authenticate the digest.

It's worth noting that the specific implementation of these steps does not adhere to a rigid pattern. Some protocols may deviate from the conventional challenge-response structure and opt for a non-interactive approach. In such cases, the Verifier does not transmit a challenge; instead, the Prover periodically dispatches attestations to the Verifier.

Chapter 3

Security of virtualized environments

3.1 Virtualization basics

Virtualization has its origins around the 60s to share very expensive hardware resources, very common for example in mainframes. Initially the servers were installed in companies and then moved to data centers for greater efficiency and maintainability. This change showed that most of these servers were unused thus highlighting a huge waste of resources. This was due to the lack of an OS that provided fundamental properties to be able to run different apps in an isolated and secure way on the same server. Another problem was the increase in CPU cores in the face of single threaded applications which caused a large number of servers to be underutilized by consuming so much energy power. During the 70s and 80s, attention was lost on virtualization thanks to the diffusion of cheap mini and personal computers. Again in the 90s it was seen as a new way to address needs such as reliability. Currently the main focus is on x86 architectures.

3.1.1 How virtualization works

To introduce virtualization it is good to define which are the main components involved: Virtual Machine i.e. software emulation of a physical machine running an operating system and some applications; Host OS that is Operating system running on the physical machine that is responsible for virtualizing the underlying hardware. This level also includes the Hypervisor; Guest OS: operating system emulated in the above VM that is unaware that it is running in a virtualized environment; Hypervisor - software that performs hardware resource virtualization processes.

Hardware virtualization allows you to define the characteristics of the hardware of the emulated machine and available to the Guest OS. To date, various solutions are used for CPU virtualization: Trap and Emulate, Para-virtualization which obtains excellent performance but little compatibility with the OS, Dynamic Binary

translation which is the best in terms of compatibility but is slow, Hardware assisted which is the most used nowadays.

Memory paging is currently used to access memory, which allows access to physical memory locations starting from a virtual memory address. For virtualization you need another layer to have virtualized memory addresses in the Guest OS mapped to virtual addresses in the Host OS and finally translated into physical addresses. To avoid this double translation, the Shadow Page Table is used which keeps track of these mappings.

Regarding I/O virtualization, however, several solutions are possible: Device emulation which is slow and computationally expensive while offering great flexibility, Para-virtualization which offers flexibility and greater speed while being computationally expensive and finally Direct assignment which is the fastest and least expensive among the solutions but requires an exclusive allocation of the device by performing a sort of passthrough.

3.1.2 Advantages and limitations

Enabling virtualization provides several benefits. Among them we can find isolation, flexibility and agility. Isolation refers to running critical applications in isolated and different OSes, this ensures that a malicious or malfunctioning service does not compromise other services. Flexibility and agility are obtained thanks to the full control of the VMs or the possibility to switch them off, on again and migrate them. Even the deployment of new servers is simplified and more immediate.

In addition to the advantages, however, there are also some limitations for example there will be an additional overhead running the same application as the guest OS must also be running in addition to it, nevertheless this overhead can be considered acceptable for most applications to execute. Another limitation is having to manage heterogeneous hardware for all running applications.

3.1.3 Enabling tools

As previously discussed, the creation of virtual machines hinges on the utilization of Hypervisors, which primarily adhere to two distinct architectural types: Type 1 and Type 2.

These two architectures diverge based on distinct design principles, namely the optimization of performance and the ease of virtual machine creation and utilization.

Type 1 hypervisors operate directly on the bare metal, constituting the sole layer that interfaces with the underlying hardware. While they offer superior performance, their installation and deployment procedures tend to be more intricate. Noteworthy implementations of Type 1 hypervisors include VMware ESXi, Xen, and Microsoft Hyper-V.

Conversely, Type 2 hypervisors operate within the confines of the Host OS, akin to standard applications. They boast straightforward procedures for VM installation and creation, albeit at the cost of slightly diminished performance compared to

Type 1 counterparts. Prominent examples of Type 2 hypervisors encompass Oracle VirtualBox and VMware Workstation.

A third hybrid type is also in existence, wherein a hypervisor is integrated into the Host OS kernel, facilitating uncomplicated installation and deployment. KVM exemplifies this hybrid hypervisor implementation. In contemporary Linux environments, KVM is extensively utilized in conjunction with QEMU, a machine emulator and virtualizer. KVM functions as a loadable kernel module, effectively transforming Linux into a dynamic hypervisor. It harnesses hardware-assisted virtualization via the Linux kernel, incorporating vital components such as the memory manager, process scheduler, and I/O stack.

3.2 Advent of Cloud Computing

Since 2006, Cloud Computing, a technological paradigm that enables the provision of IT resources on demand via the Internet, has spread widely in companies. Cloud Computing is described by NIST [15] as a model due to its practical characteristics, namely: on-demand self-service; ubiquitous network access; location independent resource pooling; rapid elasticity; measured Service.

3.2.1 Cloud Computing models

Again, NIST has proposed models for Cloud Computing [15] based on the level of flexibility, control, and management offered. There are three main models and they are:

- **Software as a Service (SaaS)**: a software distribution model in which an entire application is deployed as a cloud service. Both the application and the infrastructure it leverages are hosted on the chosen cloud service and fully maintained and updated by them.
- **Infrastructure as a Service (IaaS)**: hardware infrastructure delivery model in which the cloud provider has the task of providing the user with computation, storage, networking and virtualization capacity on demand via the Internet.
- **Platform as a Service (PaaS)**: this cloud model offers users a scalable and flexible cloud platform to cater for every possible type of app operation such as develop, deploy, run and manage. Developers don't need to configure and maintain the operating system and hardware. Applications built directly with this model are quickly scalable providing the user with greater focus on the application code.

3.2.2 Advantages and limitations

The widespread adoption of this paradigm can be attributed to the manifold advantages it brings to the forefront:

1. Elasticity: One of the prime benefits lies in the seamless scalability that this approach offers. Systems can be effortlessly scaled up or down to meet fluctuating demands, ensuring optimal resource utilization while avoiding overprovisioning.

2. Continuity and Reliability: Services remain consistently operational regardless of time and location. This unwavering accessibility ensures a dependable experience for users, bolstering the reputation and trustworthiness of the services provided.

3. Infrastructure Management Simplification: Embracing this paradigm eliminates the need for intricate infrastructure management. This, in turn, allows stakeholders to focus more on their core objectives and mission, as the nitty-gritty of infrastructure details become secondary.

4. Timely Software Updates: With this approach, the release of software updates becomes instantaneous. Users can swiftly access the latest features, security enhancements, and bug fixes without enduring prolonged downtime or manual interventions.

5. Unlimited Resource Potential: Another notable advantage is the potential for unlimited resources. Cloud environments can be readily provisioned with the necessary resources, ensuring that applications and services operate optimally even during peak loads.

6. Redundancy and Data Reliability: The paradigm's architecture often incorporates redundancy, ensuring that systems have backup mechanisms in place. This bolsters data reliability and system resilience, minimizing the risk of catastrophic failures and data loss.

7. Economic Considerations: From an economic perspective, the model offers significant advantages. Users only incur costs for actual usage and requirements, sidestepping the inefficiencies of traditional hardware resource wastage.

8. Innovation Acceleration: By offloading infrastructure management, organizations gain the bandwidth to focus on innovation and the development of their core offerings. This accelerates the pace of technological advancement and competitiveness.

9. Global Accessibility: Cloud-based services are accessible from virtually anywhere, promoting collaboration and allowing users to engage with resources without geographical limitations.

10. Environmental Impact: The cloud paradigm can also have positive environmental implications, as it enables more efficient resource allocation, reducing energy consumption and minimizing the carbon footprint associated with traditional data centers.

Collectively, these advantages underscore the transformative potential of the cloud computing paradigm. By harnessing its capabilities, organizations can achieve

enhanced operational efficiency, rapid innovation, and a more agile response to evolving market dynamics.

Conversely, it's important to acknowledge the substantial limitations that exert a considerable influence on two fundamental domains: data handling and network connectivity.

Turning to data, a noteworthy shift occurs as users cede a degree of control over the data they generate. The repository transitions to the cloud, prompting a nuanced concern surrounding the clear delineation of legal ownership for such data. This provokes contemplation regarding the potential vulnerability of these data sets to scrutiny by external entities, either for investigative purposes or comprehensive analysis.

Shifting our focus to connectivity, the constraints manifest in a more pronounced manner. When severed from an internet umbilical cord, the accessibility to applications and data becomes an unattainable commodity. This underscores the paradigm's dependency on an unfaltering internet connection. Consequently, it's discernible that the cloud model isn't optimally aligned with applications that hinge on minimal latency, indispensability during internet downtimes, or the generation of voluminous data streams.

In conclusion, although the cloud offers many benefits, its limitations are woven into how it works, requiring careful consideration of its suitability for different uses.

3.3 Cloud Security

Although cloud computing is undoubtedly a promising concept, the issue of cloud security presents a notable challenge. It's important to recognize that every security concern that plagues traditional computer systems persists within the cloud environment, necessitating targeted measures to address these issues.

3.3.1 Threats and Vulnerabilities

In the realm of the Cloud, a threat materializes as a conceivable event, either with malicious intent or stemming from incidental causes, which has the capacity to jeopardize the integrity of Cloud resources. The Cloud Security Alliance (CSA) [16] has pinpointed a range of cloud threats that warrant attention and proactive mitigation strategies. These threats underscore the organization's commitment to enhancing the security posture of cloud environments.

Diverse Service Delivery and Receiving Models

In the landscape of cloud computing, where off-site external service providers extend servers, storage, and applications, organizations face the imperative of evaluating potential risks associated with surrendering control over their infrastructure. Moreover, as data crosses geographical borders, it becomes subject to an array of federal laws. A proactive approach involves implementing robust end-to-end encryption

and adopting appropriate trust management mechanisms, which can serve to mitigate a portion of these risks.

Abuse and Malicious Exploitation

Cloud is renowned for provisioning significant resources such as bandwidth and storage capacities. While certain vendors offer predefined trial periods for their services, they often lack adequate control over potential attackers, malicious users, or spammers who might exploit these trials. This vulnerability can potentially provide a gateway for intruders to orchestrate malicious attacks, transforming the platform into a breeding ground for severe security breaches. Vulnerable areas span from password and key cracking to the launch of dynamic attack vectors, Distributed Denial of Service (DDoS) assaults, and the operation of Captcha solving farms, among others. To fortify against such perils, adopting proper validation and verification during initial registration, reinforced by robust authentication mechanisms, is imperative. Additionally, a comprehensive network traffic monitoring strategy should be implemented to ensure ongoing security vigilance.

Insecure Interfaces and APIs

Cloud providers frequently release a suite of interfaces and APIs to empower customers in crafting interfaces for seamless interaction with Cloud services. These interfaces often introduce an additional layer atop the foundational framework, inadvertently adding complexity to the Cloud architecture. Regrettably, these interfaces can provide a conduit for vulnerabilities present in the existing API to infiltrate the Cloud environment. Mishandling such interfaces can usher in threats like clear-text authentication, unprotected content transmission, and improper authorizations. Mitigation involves the deployment of a robust security model for the Cloud provider's interface. This includes implementing robust authentication and access control mechanisms accompanied by encrypted data transmission.

Malicious Insiders

One of the most significant and concerning threats within the cloud computing landscape revolves around malicious insiders. Organizations often fall short in disclosing comprehensive information about their employee hiring procedures and the extent of access granted to internal resources. This threat finds its roots in the absence of transparency between IT services and customers functioning under a singular management domain. Consequently, a scenario arises where an employee gains unauthorized elevated access, inevitably leading to the breach of data confidentiality and service integrity. Furthermore, this creates an environment wherein an insider attacker can exploit vulnerabilities to gain access to confidential data and undermine cloud services. Such an intrusion may occur through an attacker seamlessly infiltrating the system, possibly bypassing firewalls or intrusion detection systems by mimicking legitimate activities. Safeguarding against malicious insiders necessitates enhanced transparency, rigorous access controls, and vigilant

monitoring mechanisms, all pivotal in preserving the integrity and security of cloud environments.

Shared Technology Challenges in Multi-Tenancy Environments

In the context of multi-tenant architecture, virtualization emerges as a cornerstone for offering shared, on-demand services. Within this framework, a single application is made accessible to diverse users through the same virtual machine. However, vulnerabilities within a hypervisor can be exploited by malicious actors, granting them unauthorized access and control over legitimate users' virtual machines. This form of threat reverberates most notably in Infrastructure as a Service (IaaS) setups, where shared resources might not be inherently designed to deliver robust isolation for multi-tenant configurations. Effectively mitigating this challenge demands the implementation of Service Level Agreements (SLAs) for consistent patching, the establishment of strong authentication mechanisms, and stringent access controls governing administrative tasks.

Data Loss and Leakage

In the dynamic and shared environment of the Cloud, data vulnerabilities leading to compromise or theft can manifest through multiple avenues. Threats in this category encompass inadequate authentication, authorization, and audit controls; employment of feeble encryption algorithms and keys; susceptibility to unauthorized association; reliability concerns stemming from data centers; and the absence of robust disaster recovery mechanisms. Addressing these challenges necessitates a range of strategic solutions. These encompass enhancing the security of Application Programming Interfaces (APIs), ensuring data integrity through secure storage mechanisms for essential keys, and implementing efficient data backup strategies.

Service or account hijacking

It involves a process wherein a client is led to a malicious website. This manipulation is often achieved through deceptive practices like fraud, phishing, and exploiting software vulnerabilities. Frequently, attacks of this nature stem from the reuse of compromised credentials and passwords. In the cloud computing context, if an attacker gains access to someone's credentials, they can compromise activities, manipulate transaction data, disseminate falsified information, or reroute clients to unauthorized sites while infiltrating the hacked account. To counter this threat, several mitigation strategies come into play. These encompass the implementation of robust security policies, the enforcement of stringent authentication mechanisms, and the vigilant monitoring of user activities.

Risk Profiling

Cloud solutions liberate organizations from the burdens of hardware and software ownership and upkeep, yielding substantial benefits. However, this reduced involvement can render them oblivious to internal security protocols, security compliance

measures, hardening practices, patching procedures, as well as auditing and logging processes, thereby elevating the overall risk profile of the organization. To navigate this challenge, cloud providers should consider divulging partial details about the underlying infrastructure, logs, and data access. Furthermore, the establishment of a vigilant monitoring and alerting system is paramount to maintaining a robust security stance.

Identity Theft

Identity theft encompasses a fraudulent practice wherein an individual assumes another person's identity to gain access to resources or illicitly secure credit and other advantages. The repercussions for the victim of identity theft are substantial, potentially leading to adverse consequences and losses, including being wrongly held accountable for the perpetrator's illicit actions. This threat often arises due to vulnerabilities such as weak password recovery methods, phishing attacks, and keyloggers, among other tactics. A robust solution involves the implementation of formidable multi-tier authentication mechanisms, as well as the adoption of robust password recovery processes. These measures act as a bulwark against the exploitation of identity and contribute to a more secure digital landscape.

We can consider cloud vulnerability as the weakness or loopholes in the security architecture of Cloud, which can be exploited by an adversary via sophisticated techniques to gain access to the network and other infrastructure resources. On the other hand, a cloud attack can be envisioned as a malicious endeavor aimed at harvesting, disrupting, denying, degrading, or erasing cloud resources.

3.3.2 Main issues

A Security Issue can be comprehended as an encompassing term employed to denote occurrences such as events, actions, misconfigurations in software or hardware, or vulnerabilities in applications. These anomalies deviate from the expected security context. Incidents that transpire across assets encompassing attacks, misconfigurations, faults, damages, loopholes, and system weaknesses.

Data storage and Un-trusted computing

Data stored in the cloud remains isolated and opaque to customers. Customers, on the other hand, either hesitate to share their information or constantly grapple with the fear of data falling into the wrong hands, along with potential unfavorable consequences that can arise during manipulations and processing. Ensuring their data's consistency during computation, maintaining confidentiality at every processing stage, and guaranteeing perpetual storage for record updates are crucial considerations. However, when data is remotely stored or entrusted to third-party storage, a significant challenge arises: users remain uncertain about what occurs after data is stored in the cloud. The data owner lacks awareness about the precise location of the cloud storage center, the security services employed, and the mechanisms ensuring the security of cloud data.

A crucial issue in the cloud computing model is the loss of data control; it fails to offer comprehensive control over data, making it challenging to verify data integrity and confidentiality. Cloud computing customers find themselves physically detached from their data, storage, and computing servers. The adoption of multi-location data storage introduces a novel security dimension, potentially leading to new security threats and legal complexities, as data stored across different parts of the world are subject to varying policies.

Network and internet

Network Security Challenges in the Cloud revolve around achieving robust security in a dynamically evolving network environment. The inherent dynamism of both the Cloud (with features like dynamic scaling and live migrations) and networking (such as Software-Defined Networking) significantly amplifies the complexities of network security configurations. Network security firewalls often struggle to adapt to factors like migrations, scaling, new connections, and diverse paths. Cloud infrastructures encompass not only the hardware that stores and processes data but also the intricate pathways through which data traverses. Internet, the prevalent transmission medium, inherently carries risks and challenges; its security cannot be assumed.

Whether spurred by hacktivism or cyberwarfare, Distributed Denial of Service (DDoS) attacks have become increasingly prevalent. Three distinct scenarios emerge within DDoS attacks in cloud environments:

- DDoS attack targeting servers: Overloading servers either through processing a single malicious request exploiting vulnerabilities or managing an onslaught of numerous requests.
- DDoS attack affecting the network: The network link becomes fully saturated with counterfeit requests from the attack, causing it to reach its bandwidth capacity and impeding legitimate connections.
- DDoS attack targeting intermediate network devices: One or more intermediate network devices, like routers, DNS resolvers, or load balancers, can also be overwhelmed, leading to a considerable processing load and saturation of bit rates per second.

Cryptography

Cryptographic methods often stand as the initial line of defense in bolstering security measures. Nonetheless, their implementation and design necessitate meticulous attention, as cryptography alone cannot assure comprehensive security. Potential vulnerabilities encompass bugs, the use of insecure or outdated cryptographic algorithms, inadequate password selection, and subpar key management. Notably, the rise of brute-force attacks poses a significant menace due to their relatively simpler execution. Two fundamental factors contribute to the intensification of this threat:

the continual evolution of technology and the refinement of password cracking techniques. Modern computers harness substantial processing power, distributed across diverse platforms, including multi-core CPUs and high-clock-rate Graphics Processing Units (GPUs). This computational prowess facilitates rapid exploration of vast combinatorial keyspaces encompassing upper- and lower-case letters, digits, and symbols. Beyond robust hardware, malicious actors rely on refined methodologies honed over time, enabling efficient navigation of the keyspace universe in terms of algorithmic complexity. A series of substantial database password breaches, spanning numerous years, has afforded insight into user habits surrounding password selection. These breaches have also supplied the foundation for the construction of extensive rainbow tables and colossal dictionary lists, often reaching into the hundreds of millions in scale.

Virtualization

Cloud service development for business purposes necessitates a foundation of trust in virtual machines (VMs) by the cloud provider. In this context, virtualization stands as a cornerstone requirement for any service. While the concept of multi-tenancy and virtualization holds promising profitability, it is not immune to threats and attacks. The dynamic nature of the cloud empowers providers to craft, modify, and replicate VM images. Administrators face the risk of hosting and disseminating malicious images, introducing potential vulnerabilities. The cloud provider risks data leaks if inadvertently publishing images, as these images contain fully configured applications and data. Cloud users face the hazard of running vulnerable, malicious, outdated, or unlicensed images stored within an insecure or improperly administered repository.

Moreover, within the Infrastructure as a Service (IaaS) architecture, VMs, though subject to the control of customers, introduce potential vulnerabilities. VM hopping involves maliciously accessing different VMs owned by other customers by exploiting VM-to-VM or VM-to-VMM attack vectors and successful attacks yield a range of security issues, allowing unauthorized monitoring of resource utilization, system configuration alterations, file manipulation, and potential data leaks.

Authentication, Authorization, Anonymization

These stand as vital pillars for ensuring secure and privacy-preserving operations. Authentication safeguards access to cloud applications, authorization controls permissions, and anonymization conceals sensitive identity information, collectively enhancing the reliability and confidentiality of cloud services.

Authentication techniques are integral to establishing secure access to cloud applications. Weak authentication mechanisms can expose the cloud to attacks like brute-force and dictionary attacks. Cloud service providers furnish various authentication methods and options for credentials recovery.

Authorization, on the other hand, involves the process of granting or denying permissions to individuals or systems. In the cloud, implementing authorization mechanisms for each user and data becomes intricate due to the vast number of

users and data points. This complexity can lead to security implications, such as data leakage. Inadequate or flawed authorization checks could potentially allow unauthorized third-party access to cloud-based services.

Anonymization systems play an important role in preserving the privacy of cloud customers. Concealing and safeguarding the identity information of cloud users is paramount. The cloud offers a spectrum of anonymization techniques to ensure privacy, empowering users to interact with cloud services without revealing sensitive identity-related data.

3.3.3 Design principles

In the context of cloud computing, ensuring robust security measures is paramount to safeguarding sensitive data and operations. **Security by design** entails a proactive approach, integrating security principles into the very fabric of the cloud architecture. The following dive into key security by design principles that contribute to the establishment of a resilient and fortified cloud environment:

1. Least Privilege: This principle revolves around granting the minimum necessary privileges to users or processes, restricting access to only what is required for their legitimate functions. By adhering to the least privilege principle, potential avenues for unauthorized access and data exposure are minimized.

2. Separation of Duties: Separation of duties enforces a system where no single individual possesses all the necessary permissions to carry out a critical operation. This mitigates the risk of an individual abusing their access for malicious purposes.

3. Separation of Privilege: Similar to separation of duties, separation of privilege ensures that multiple authorization factors are needed to execute sensitive actions. This adds an extra layer of security, making it more challenging for attackers to breach the system.

4. Defense in Depth: This multifaceted approach involves layering security measures throughout the system, from the perimeter to the core. Even if one layer is compromised, others remain intact, effectively fortifying the overall security posture.

5. Fail Secure: A "fail secure" stance ensures that a system defaults to a secure state in case of failure or breach. This principle guards against potential vulnerabilities that could arise from system failures.

6. Complete Mediation: Complete mediation mandates that every access to resources is validated and authorized. This ongoing validation ensures that no unauthorized actions slip through the cracks.

7. Psychological Acceptability: This principle recognizes that security measures should be designed in a way that users find them acceptable and conducive to their workflow. If security measures are too cumbersome, users might bypass them, inadvertently introducing vulnerabilities.

8. Weakest Link: The weakest link principle emphasizes that the overall security of a system is only as strong as its weakest component. Thus, addressing vulnerabilities in every facet of the system is imperative to prevent potential breaches.

Another efficient approach that shape security landscape is **Zero Trust**. This approach is grounded in the notion of never automatically trusting any entity, whether internal or external to the network. Instead, verification is the foundation, followed by the principle of Least Privilege and Default Deny. Additionally, full visibility and inspection is maintained to monitor and scrutinize all network traffic, enabling prompt detection of anomalies or potential threats. The realization of Zero Trust can also be accomplished by implementing what is commonly referred to as Distributed Security. This entails the deployment of diverse protective measures across the cloud infrastructure. This encompasses the utilization of firewalls, virtual private networks (VPNs), and the establishment of robust password policies. These mechanisms work accordingly to collectively fortify the protection of data and communications against unauthorized access and potential breaches.

By integrating these security by design principles into the cloud computing framework, organizations can establish a comprehensive security posture that proactively addresses potential threats, fostering a more secure and resilient cloud environment.

3.4 TEE in Cloud Computing

To address these security challenges, the concept of Trusted Execution Environment (TEE) [17] emerges as one of fundamentals building blocks. TEEs guarantee a secure and isolated environment within a computing system where sensitive operations can be executed with the highest level of trust and confidentiality. TEE establishes a secure enclave, shielded from the external environment, and guarantees that the code and data within it remain unaltered and protected from unauthorized access. As cloud computing embraces the TEE concept, its implementation becomes crucial for ensuring the robust security posture of cloud services and applications. Several Trusted Execution Environment (TEE) technologies are relevant for enhancing security in cloud computing environments.

3.4.1 Intel SGX

Introduced by Intel in 2015 as part of the 6th Generation Core processors, SGX (Software Guard Extensions) was designed to provide robust protection against memory bus snooping and cold boot attacks. Its primary objective is to empower developers to partition their applications, thus affording them the ability to safeguard specific sections of code and data within isolated enclaves. Within the enclave's confines, data and code are shielded with a heightened level of security. Only authorized code can gain access to the memory within these enclaves. One of SGX's key features is its utilization of hardware-based memory encryption. Unauthorized

attempts to access or tamper with this enclave's memory trigger exceptions, enhancing the system's overall security. SGX significantly expands the Intel's Instruction Set Architecture (ISA) with the incorporation of 18 new instructions. Furthermore, it introduces the novel capability of securely offloading computations to environments where the underlying components, including the hosting application, host kernel, System Management Mode (SMM), and peripheral devices, are considered untrustworthy.

From the official paper [18], SGX introduces novel data structures to uphold and enforce the security properties inherent to SGX. The key components of these structures are:

- Enclave Page Cache (EPC): protected memory used to store enclave pages and SGX structures, divided into 4KB chunks EPC page. EPC pages can either be valid or invalid;
- Enclave Pache Cache Map (EPCM): safeguarded construct utilized by the processor to monitor the contents of the Enclave Page Cache (EPC). This structure comprises a sequence of entries, precisely corresponding to each page within the EPC. Managed exclusively by the processor as an integral element of various SGX instructions, the EPCM remains beyond the direct reach of software or devices.
- SGX Enclave Control State (SECS): each enclave instance has an enclave control structure which contains specific Meta data;

3.4.2 AMD SEV

Secure Encrypted Virtualization (SEV) [19] introduces a critical feature within AMD EPYC processors, serving as a safeguard for sensitive data in a cloud context. Its core function is to shield Virtual Machines (VMs) from potential breaches by privileged software or administrators in multi-tenant cloud environments. SEV relies on the synergy of AMD Secure Memory Encryption (SME) and AMD Virtualization (AMD-V) to establish cryptographic isolation between VMs and the hypervisor.

Each VM is granted an individual ephemeral Advanced Encryption Standard (AES) key, playing an important role in the encryption of memory during runtime. This AES key collaborates with the on-die memory controller's AES engine to encrypt and decrypt data transfers between the main memory and VMs. Notably, the AMD Platform Security Processor (PSP), a 32-bit Arm Cortex-A5 micro-controller embedded within the AMD System-on-Chip (SoC), takes charge of managing these unique per-VM keys.

3.4.3 Intel TDX

Intel Trust Domain Extensions (TDX) [20] emerges as a novel architectural enhancement within the 4th Generation Intel Xeon Scalable Processor, introducing

robust support for confidential computing. TDX facilitates the deployment of virtual machines within the Secure-Arbitration Mode (SEAM), featuring encrypted CPU state and memory, integrity preservation, and remote attestation. The primary objective of TDX is to enforce enhanced isolation for virtual machines through hardware-assisted mechanisms, while simultaneously curtailing the exposed attack surface concerning host platforms. TDX draws upon an amalgamation of established Intel technologies, incorporating Virtualization Technology (VT), a set of hardware-assisted virtualization features in Intel processors, alongside Total Memory Encryption (TME) and Multi Key Total Memory Encryption (MKTME), to protect against attackers who have physical access to a computer's memory and attempt to steal data, in addition to SGX mentioned before.

Chapter 4

Deep Attestation

Having introduced the concepts of Remote Attestation and the fundamentals of cloud computing, let us now examine how these two overarching areas tend to merge when discussing Deep Attestation: attestation of the VM involved in the process and the subsequent attestation of the underlying virtualization layers and corresponding trusted platforms.

4.1 Remote attestation of virtualized systems

The introduction of virtualization brings forth new attestation challenges, primarily stemming from the software-based nature of TPM and the inherent flexibility offered by virtualization. Diverse range of challenges exists which will be analyzed below.

Protecting vTPM Storage and Secrets

In a trusted platform that is not virtualized, the TPM serves as a safeguard for protected storage and the execution of operations on this protected data. This design effectively prevents attacks on the platform's software from compromising the security of these protected secrets. This security is achieved because the TPM operates within a distinct address space that is isolated from direct access by the platform's software. However, when it comes to a virtualized TPM (vTPM), a different set of threats emerges that require careful consideration. These threats extend to the security of the vTPM's stored data, especially in scenarios involving migration and reboots. TPMs, including vTPMs, maintain certain states that are expected to persist even through system reboots, i.e. an Endorsement private key, while other states are intended to be reset, i.e. PCRs.

Layer binding

After successfully attesting a VM, a Remote Challenger may have an interest in verifying the trustworthiness of the layers situated beneath the VM. This verification aims to ensure that these underlying layers do not tamper with the VM's behavior

or attestation reporting. It's important to note that the N-1 layer, where the highest layer is designated as Layer N, and subsequent layers are denoted as N-1, N-2, and so forth, may also operate atop a virtualized trusted platform. Consequently, the Remote Challenger may need to conduct multiple rounds of attestation for virtualization layers until reaching the bottom-most layer, known as Layer-1, which operates directly on a physical trusted platform.

One potential attack against a virtualized trusted system involves deceiving the Challenger by making it believe that it's conducting a deep attestation of layers on a single physical platform while redirecting the attestation requests to other systems. Similarly, another attack vector could involve the N-1 layer acting as an active man-in-the-middle, forwarding attestation evidence from another N-1 layer in response to attestation requests.

Supporting Different vTPM version

Another potential challenge that virtualized trusted platform implementations may encounter involves accommodating vTPMs that adhere to a different version of the TPM specification compared to the pTPM connected to the platform. Similarly, managing multiple VMs, each of which anticipates having a vTPM compliant with a distinct specification from the vTPMs associated with other VMs, poses a similar challenge.

Migration

One significant obstacle encountered in virtualized trusted platform implementations pertains to facilitating migration while upholding security standards. Historically, trusted platform roots, like the TPM, were tightly associated with the hardware platform, making them difficult to transfer to different systems. In a virtualized environment, the trusted roots of VMs might not possess a robust cryptographic link to the underlying hardware roots. This lack of a strong connection could potentially open the door to unauthorized actions such as duplicating, migrating, or deleting the virtual TPM. It is imperative that the virtual trusted platform implements measures to prevent these unauthorized actions on virtual trusted roots.

4.1.1 Attestation components and virtual TPM

In order to maintain uniformity across the virtualization architectures, which are elaborated further in the low-level specifications, several requirements must be outlined. These components can be organized into three distinct areas:

- The physical platform: it includes the fundamental capabilities offered by the lowest-level platform, typically implemented in hardware and firmware. These capabilities are less prone to change compared to higher layers within the platform. In virtualized systems, the physical platform is shared among various layers of the reference architecture, including logically isolated VMs.

In some instances, specialized hardware or firmware ensures the isolation of VM accesses, while in others, this capability is provided by the Virtual Machine Monitor. Certain virtualization models, such as para-virtualization, allow VMs to directly access hardware devices to enhance performance, especially when only one VM is granted access to the device. In this context, a conventional setup usually incorporates a RTS and a RTR, both residing within a TPM chip on the motherboard. However, when it comes to virtualization, one approach for providing a TPM to each VM instance is to utilize a software-based version of it. As we'll discuss in the following section, this approach is not a secure one, as it introduces the risk of potential exploitation due to its software-based nature. Additionally, the issue of identification and, consequently, credentials becomes a consideration. Each TPM is linked to a specific collection of credentials, typically X.509 certificates, which serve the purpose of asserting various attributes of the TPM. These attributes are securely linked to knowledge of a confidential private key stored within the TPM through the certificate issuer's signature.

- The virtual machine monitor: it covers the components at the VMM level and privileged VMs. The primary role of this level is to maintain a set of measurements reflecting the VMM's operational state. This allows a Remote Challenger to attest to the entire platform's state once the VMM has booted and established a transitive trust chain. The vPlatform Manager handles the full lifecycle management of instantiated vPlatforms for servicing VMs. The vTPM is responsible for providing a dedicated TPM interface and feature set compliant with TPM specifications to a specific VM. This ensures that the VM can operate consistently, whether it's virtualized or running directly on a physical TPM. It's crucial that the vTPM architecture prevents the VM from directly accessing the internal state of the vTPM, regardless of any malicious software running within the VM.

- The virtual machine: this area contains only OS-related components, requiring no virtualization-specific modifications. Code in the VM relies on a physical platform with trusted roots. The OS's trusted roots measure subsequent code, ensuring a trust chain. To support remote attestation, the OS must have a service using a recognized attestation protocol for the Remote Challenger, which does not necessarily have to be specific to virtualization, but the attestation protocol may include virtualization-specific features for the Attestation Agent to adopt.

4.1.2 Attacks against virtual TPM

In this context, where software instances frequently substitute for their physical equivalents, the assurance of system integrity hinges on the reliability of these virtual components. Nevertheless, vulnerabilities persist such as in-memory attacks and side-channel attacks. These concerns gain prominence when considering virtualized Trusted Platform Modules, which lack the innate security attributes found in their physical counterparts. This underscores the importance of protecting these virtualized elements from potential exploits.

In-memory attack

In-memory attacks, which occur within a system's volatile memory, often target trusted applications or operating systems. By infiltrating and manipulating these applications and systems, they tend to evade suspicion and can initiate malicious processes. These attacks still require a threat actor to gain access to an environment, often through tactics like phishing, but the threat actor need not install any code, making them exceptionally difficult to detect.

Various techniques are employed by threat actors to carry out in-memory attacks, including, but not limited to:

- Shellcode Injection: This well-established technique follows a straightforward four-step process. It commences by opening a target process, which then allocates a segment of memory within the process for writing a shellcode payload into the allocated region. Subsequently, it spawns a new thread within the remote process to execute the shellcode.
- Process Hollowing: Frequently used to evade detection by security products and hunters, this technique involves creating a suspended process, removing the original executable from the process, injecting a new payload into the process, redirecting the execution to the new payload, and then initiating execution through a callback.

These sophisticated methods allow threat actors to manipulate in-memory components discreetly, making them particularly challenging to identify and counteract.

Side-channel attack

A side-channel attack involves exploiting additional information that can be acquired due to the inherent implementation of a computer protocol or algorithm. Unlike attacks stemming from protocol or algorithm design flaws or minor implementation errors, side-channel attacks leverage sources of information such as timing data, power consumption, electromagnetic emissions, and sound. Due to its cost-effectiveness, minimal computing resource requirements, and relative simplicity, side-channel attacks have been on the rise. Consequently, ensuring security within a system has become increasingly challenging. Here are two examples of side-channel attacks:

- Timing Attack: In a timing attack, an attacker measures the time it takes for a system to perform certain operations. By analyzing variations in execution times, an attacker can infer valuable information about the cryptographic keys or data being processed. For example, an attacker might exploit subtle timing differences when decrypting data to determine the encryption key used.
- Power Analysis Attack: Power analysis attacks involve monitoring the power consumption of a device during cryptographic operations. The power consumption patterns can reveal information about the operations being performed. By carefully analyzing power fluctuations, an attacker can deduce

secret keys or sensitive data. For instance, smart cards and hardware security modules are susceptible to power analysis attacks when cryptographic operations are executed.

A side-channel attack involves exploiting additional information that can be acquired due to the inherent implementation of a computer protocol or algorithm. Unlike attacks stemming from protocol or algorithm design flaws or minor implementation errors, side-channel attacks leverage sources of information such as timing data, power consumption, electromagnetic emissions, and sound. Due to its cost-effectiveness, minimal computing resource requirements, and relative simplicity, side-channel attacks have been on the rise. Consequently, ensuring security within a system has become increasingly challenging. Here are two examples of side-channel attacks:

4.2 Attestation architectures and flows

Based on the TCG Specifications [21] the attestation system offers a service to authorized Remote Challengers, enabling them to request evidence of integrity for the virtualized system. This evidence is cryptographically linked to the platform's trusted roots, such as the vTPM, and can be independently verified by remote parties.

Figure 4.1 illustrates a scenario where a Remote Challenger initiates attestation in step 1, followed by a subsequent deep attestation of the VMM through the utilization of a Deep Attestation Service.

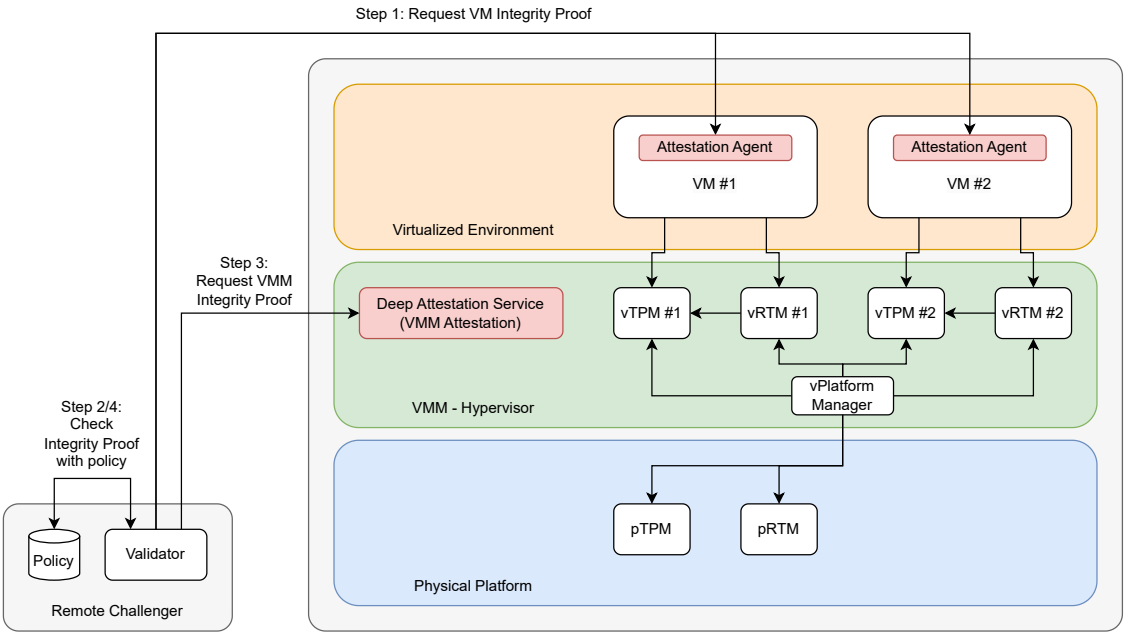

Figure 4.1. Deep Attestation Flow

Initially, the Remote Challenger determines the necessity of attestation to assess the trustworthiness of the remote platform intended for a specific purpose.

The Remote Challenger establishes a security-protected session with the Attestation Agent within the VMs. At this stage, the Remote Challenger operates under the assumption that the remote platform is not virtualized, behaving similarly to attesting a non-virtualized system.

The Attestation Agent actively listens on the network for attestation requests. Following an initial handshake, which serves to establish and authenticate the identities of the parties involved in the session, the Attestation Agent should support a privacy policy check to ascertain whether the challenging party possesses the necessary authentication and authorization to access information regarding the platform. Once the session is fully established, the Remote Challenger can send requests for integrity evidence pertaining to the remote virtualized platform.

Upon determining that it trusts the content of the VM to perform the intended service, the Remote Challenger must attest the VMM or the N-1 layer. The architecture supports two attestation models, each with distinct security characteristics.

The first model utilizes the existing attestation session between the Remote Challenger and the Attestation Agent within the VM. Through this session, the Remote Challenger can attempt to inquire about the components operating in the N-1 layer, following the attestation protocol's rules. If the Attestation Agent can provide information regarding the N-1 layer, the Remote Challenger can request the necessary data to evaluate its trustworthiness. If the Attestation Agent cannot support this operation, it returns an appropriate error, indicating that the second model should be pursued.

In the second model, the Remote Challenger establishes a separate attestation connection with the VMM Attestation component to initiate the attestation of the N-1 layer. This attestation process largely mirrors that of the VM, including the verification of security requirements for the session. The primary distinction between VM and VMM attestations lies in the specific types of components requested and, in this case, the manner in which the Deep Attestation Service communicates with the VMM to acquire the necessary information. It's important to note that the core attestation protocol remains consistent, whether applied to the VM or VMM layer. Additionally, the VMM Service VM is considered an integral part of the VMM for the purposes of this attestation, and its integrity status is also reported.

4.2.1 Layer Bindings

Methods enabling deep attestation may incorporate attestations from both virtual and physical Trusted Platform Modules. As stated by TCG:

> One of the largest challenges with performing a deep attestation of a virtualized system is how to know that the integrity evidence being presented is actually coming from the same physical system.

For every individual attestation of a virtual machine within a virtualized system, it is imperative to establish the relationship between different layers. This connection must rely on verifiable information provided by the device.

Deep attestation is considered the most robust form of VM attestation and is arguably the only meaningful way to assess a VM's trustworthiness. It's crucial to recognize that attesting to the integrity of a VM in isolation becomes meaningless if the underlying hypervisor potentially harbors malicious intent. To illustrate this, consider the attestation of a virtualized network function holding sensitive identification information for network communication. The VM's attestation might suggest that it is functioning correctly and in line with its defined software configuration, indicating full operational status. However, this assessment can be undermined by a malicious hypervisor that intercepts I/O operations, leading to the exposure of credentials or tampering with data. Therefore, it is essential to provide hypervisor attestation alongside VM attestation, creating a linkage that enables evaluators to establish their interdependency.

Currently, established methods either enable this layer binding. These solutions frequently establish this connection by associating the VM and, consequently, the vTPM with the pTPM, as discussed in the following section.

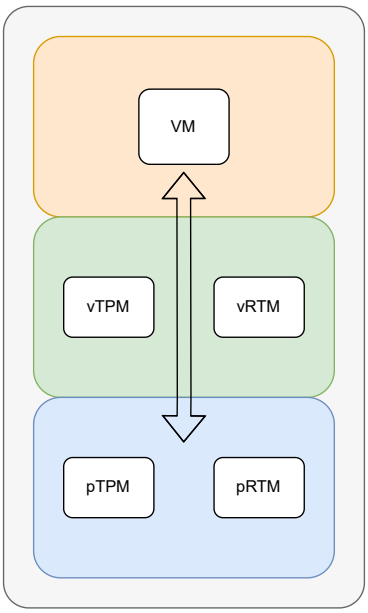

Figure 4.2. VM-VMM layer binding

4.3 pTPM - vTPM binding solutions

As previously introduced in the first chapter, there exist multiple solutions for establishing this binding, each with its own set of advantages and disadvantages that will be analyzed below.

Berger et al. [3] developed a software-based virtual TPM facility to deliver TPM functionality to virtual machines. In this setup, all virtual machines receive TPM functionality through a virtual TPM running within the management VM. In Figure 4.3 are shown the original proposed architectures.

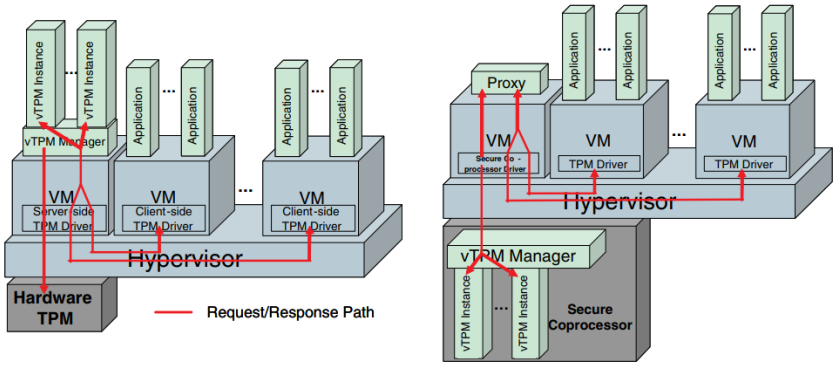

Figure 4.3. Berger et al. architectures

On the left, for the management VM itself, TPM functionality is provided by the hardware TPM, functioning in a manner similar to a system without a hypervisor, where the operating system has direct control over the hardware TPM. On the right, the architecture involves the integration of virtual TPM functionality into an external secure coprocessor card, ensuring the highest level of security for sensitive data like private keys in a tamper-responsive environment. Each virtual machine requiring TPM functionality is allocated its own vTPM instance. The vTPM manager is responsible for tasks such as creating vTPM instances and managing requests from virtual machines directed to their respective vTPM instances.

To address the layer binding problem, their solution merges the virtual TPM hosting environment with that of the virtual machine by providing two distinct views of PCR registers. The lower set of PCR registers in a vTPM reflects the values of the hardware TPM, while the upper set represents values specific to that particular vTPM.

One drawback of this method is that the management of vTPM state and keys is entirely software-based within the privileged domain, resulting in a lower level of assurance and tamper resistance compared to hardware-based solutions. The authors also proposed an alternative architecture for deep attestation. In this framework, a vTPM identity credential, specifically a vTPM EK certificate, is linked or bound to that of a pTPM, known as an AK certificate. This linkage provides hardware-level protection for the private keys generated by the vTPM instance, preventing any unauthorized manipulation by potential attackers who might gain access to the hypervisor.

Wan et al. [4] discuss alternative designs for vTPMs, focusing on a different approach rooted in para-virtualization of the pTPM [22]. This approach addresses the limitations associated with software-based vTPMs, where their operations lack hardware protection and necessitate hardware TPM virtualization capabilities.

To achieve TPM para-virtualization, modifications are required in the TPM interface. Some components of the TPM, such as PCRs and counters, need to be either replicated or partitioned for each vTPM instance.

The para-virtualization module can be integrated into the TCB. It utilizes the

hypercall interface, calls by user processes or the operating system, to expose the TPM functionality to each VM. Within the module, there's a scheduler that manages TPM access, a command filter to prevent unauthorized access to TPM functions, and a context manager responsible for maintaining associations between VMs and vTPMs, isolating TPM contexts between different VMs, and managing resource handles. Additionally, the module utilizes the TPM driver of the Virtual Machine Monitor to access the hardware TPM, while VMs use the corresponding HyperTPM driver to interact with the TPM.

In comparison to prior solutions, this approach seeks to address the security issues associated with fully virtualized TPMs when exposed to VMs, although it doesn't completely overcome the limitations inherent in the current TPM design.

In the work conducted by Orange [5], deep attestation is tackled through cryptographic techniques. The research delves into the deep attestation procedures outlined in the ETSI standards [1], specifically focusing on both single-channel and multi-channel attestation, and subsequently explain the cryptographic mechanisms proposed. Furthermore, the paper furnishes performance test results to underscore the scalability and security aspects of the proposed solution, showcasing its effectiveness in ensuring robust layer binding.

However, as already discussed in the first chapter, a drawback of this approach is its potential limitation of certain TPM functionalities, such as the ability to migrate virtual machines to an alternate hypervisor or replace TPMs.

Wang et al. [6] devised an enhanced security architecture for vTPM 2.0. Specifically, this architecture allows each VM to have its dedicated vTPM 2.0 devices equipped with TPM 2.0 capabilities.

In this solution, the critical components of vTPM 2.0, including Libtpms 2.0 (a TPM emulator) and NVRAM, which is akin to vTPM memory, are safeguarded using SGX keys and isolated within an SGX enclave.

When Libtpms 2.0 is loaded, an SGX enclave is established, and the Libtpms 2.0 program undergoes a measurement process to validate its integrity. Once the program's integrity is confirmed to be uncompromised, the code of Libtpms 2.0 is executed within the enclave's EPC. Consequently, the program enjoys protection during runtime, and only itself has access to the code within the enclave. The untrusted components of vTPM 2.0, such as the vTPM 2.0 management module, can solely invoke the functions of Libtpms 2.0 through enclave calls (ecall) and out calls (ocall).

The NVRAM of TPM is structured as a distinct file responsible for storing keys, PCR values, seeds, and other confidential data. Whenever a vTPM 2.0 device is created, a corresponding NVRAM file is generated. Because crucial vTPM data is stored in the NVRAM file, its security is of utmost importance. Consequently, SGX sealing is employed to safeguard the NVRAM. SGX offers a sealing function, allowing the preservation of sensitive data within an enclave for future utilization.

Nonetheless, it remains exposed to the same vulnerabilities as any other instance of the vTPM, despite the heightened security offered by SGX enclaves.

Chapter 5

TPM extension design

In order to improve the TPM 2.0 design and effectively handle vTPM identities and their life cycles within a host system equipped with a pTPM, Polito and HP have created an innovative solution. Provisions for the secure migration of virtual instances are also included of this extension.

Maintaining the strong hardware-level security that a pTPM offers, even in a virtualized environment, is one of the fundamental goals of this approach. The goal of this security is to secure vital identification components like seeds and primary keys, which must be hidden inside the device and should never be disclosed or left unprotected.

In this approach [2], the pTPM interface is expanded to provide the hardware-based encryption of vTPM primary keys, the creation of child objects, and hardware-based protection for vTPM primary seeds.

Binding a vTPM to a pTPM is the first step in this work, ensuring that the cryptographic keys generated within each hierarchical structure are protected by the underlying hardware platform rather than relying only on software-based security. A Storage Parent Key structure that has its roots in primary keys protects the vast majority of TPM objects, which are referred to as "ordinary objects". These primary keys, which are created from primary seeds, are the basis of each hierarchy. As a result, the solution concentrates on safeguarding the creation of vTPM seeds using the underlying pTPM and then broadens this safeguard to cover primary keys and common objects.

Seeds must be protected since they are crucial to create main keys for each tier of vTPMs. To do this, a specialized management entity called vTPM manager runs at the host level to keep track of vTPM instances and the data that goes with them, including seeds.

Following the discussion on the Storage Parent Key, the procedure for creating child objects that are fully virtualized or anchored to the physical platform is examined.

Additionally, the pTPM plays an important function in protecting the integrity and confidentiality of vTPM data by wrapping the cryptographic keys used for encrypting and decrypting the persistent state of vTPM instances throughout their life cycles.

5.1 vTPM initialisation and binding

In order to enable the hardware binding, key generation, and key utilization, certain design modifications are required in the pTPM 2.0 interface. These modifications will be triggered by the extended software of vTPM. The following sections will introduce the extensions needed to establish hardware binding, which involves introducing a new internal object into the TPM. These processes are part of the vTPM initialization phase.

To facilitate hardware linkage, key creation, and key deployment, specific alterations are essential in the pTPM 2.0 interface. These changes are initiated by the enhanced software components of vTPM. In subsequent sections, it will be outlined the necessary extensions for achieving hardware linkage, which include the addition of a novel internal entity to the TPM. These procedures fall within the initialization stage of the vTPM.

5.1.1 Virtual Primary Seed - VPS

Essential for the solution is the introduction of a new TPM object named the Virtual Primary Seed (VPS). It is conceptualized as a variant of the standard SDO. Specifically, it capitalizes on the existing keyed hash object type, combined with a unique set of object attributes bit-mask. Opting for an existing object type minimizes alterations to the specifications. Each VPS is encapsulated by a parent pTPM key, termed the Seed Wrapping Key (SWK). The hierarchy of the SWK serves to pinpoint the VPS hierarchy.

Concerning its object attributes, there are two potential strategies stemming from the SDO attribute selection. These strategies necessitate that all sign, decrypt, and restricted attributes are set to clear: setting the restricted bit, which is typically forbidden in the current specifications for an SDO and setting the sensitiveDataOrigin flag, which is normally clear in a standard SDO since the data is generated outside of the TPM.

Both approaches are viable, nevertheless the first option is chosen as it effectively prevents the pTPM from using the VPS for operations involving unknown data structures or external data blobs.

Seed generation can be done with the RNG of the pTPM. Since the seed needs to adhere to the size restriction of a standard SDO to prevent breaches of the existing constraints highlighted in specifications [7], it's crucial to recognize a limitation in our proposal. As previously mentioned, this constraint is currently set at 128 bytes, which surpasses the size of primary seeds used in prevalent TPM 2.0 simulators, usually at 64 bytes.

Seed Wrapping Key

A parent pTPM key, referred to as the Seed Wrapping Key (SWK), wraps each VPS. This SWK hierarchy serves as a means to distinguish the VPS hierarchy. The purpose of the SWK is to securely wrap the primary seed, ensuring that its context can be stored within the vTPM without revealing the seed in plaintext.

Hierarchy tree

During the instantiation of vTPM, it's necessary to create the virtual Endorsement Primary Seed (vEPS), virtual Platform Primary Seed (vPPS), and virtual Storage Primary Seed (vSPS) using the underlying pTPM. The NULL hierarchy seed can be generated either through the pTPM RNG or be fully implemented in software. In Figure 5.1, you can see the relationship between the pTPM's per-hierarchy Secret Wrap Keys (SWKs) and multiple VPS objects connected to various vTPMs.

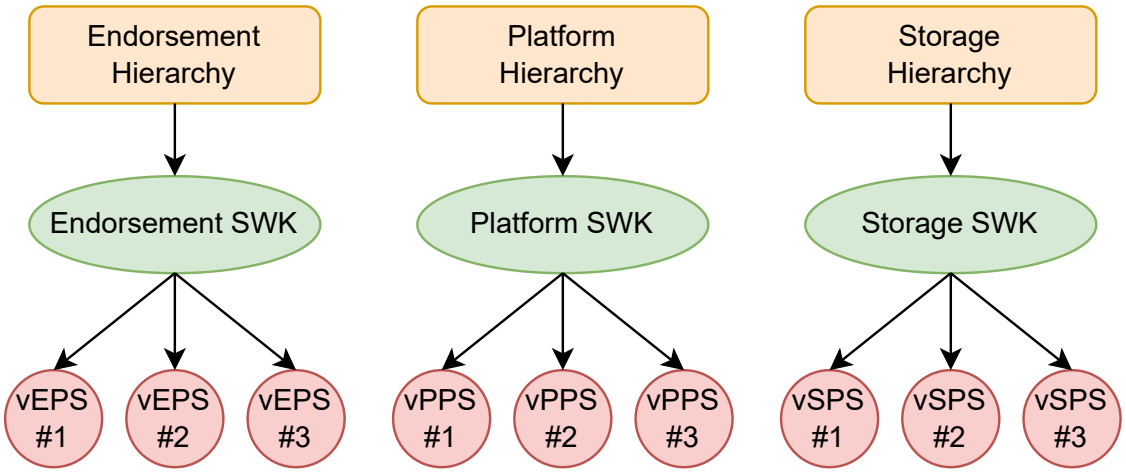

Figure 5.1. VPS hierarchies tree

During vTPM runtime, the instance is only aware of the handle for each VPS, while it's the responsibility of the pTPM to safeguard and manage access to the seed values.

5.1.2 Initialisation

During the initialization phase, the vTPM manager is the primary point of interaction. It takes the lead in establishing vTPM instances within the host platform, subsequently providing access to these instances for virtualized operations. Before commencing the creation of any vTPM instances, the manager's responsibility lies in confirming the existence of three parent keys within the pTPM or creating them if necessary. These keys pertain to the three permanent hierarchies, namely Platform, Endorsement, and Storage.

The vTPM manager should execute the following operations when initializing a new vTPM instance:

1. The vTPM manager initiates the vTPM instance by loading the software module and configuring the vTPM's handles for vEPS, vPPS, and vSPS SWK keys;

2. Subsequently, the vTPM instance receives an initialization command from the entity utilizing the vTPM;

3. The vTPM instance, in turn, requests the creation of vEPS, vPPS, and vSPS objects from the pTPM by providing the handles of the corresponding SWK keys, and it stores both their public and private components within its state;

4. The vTPM software loads the vEPS, vPPS, and vSPS objects into the pTPM and retains their handles within its state;

5. If the vTPM initialization process proves successful, the vTPM generates a primary symmetric encryption key, the vTPM State Protection Key (VSPK), under its Platform hierarchy by invoking the pTPM interface using the vPPS handle, preserving the resulting handle internally.

The VSPK, generated during the final step of vTPM initialization, is necessary for the store and restore operations of the vTPM and can be regenerated during each restoration operation since the vTPM instance should possess knowledge of the public template of the VSPK. As a result, the vTPM manager must define the template and provide it to the vTPM instance. Furthermore, the template should be designed to enable the vTPM instance to decrypt its own state exclusively. To achieve this, the unique field within the public template can be utilized to generate a distinct VSPK for each vTPM instance under the same pTPM Platform hierarchy. To safeguard the state against potential attackers, an authorization policy can be specified to limit the utilization of the VSPK.

5.2 Fully softwarised objects

When conceptualizing a vTPM it is imperative to factor in flexibility. This flexibility is vital to provide both performance and security, in respect to the level of assurance demanded by specific workloads. With this objective, it is proposed that the vTPM should offer support for the creation of two distinct types of objects: pTPM-protected child objects, leaving their sensitive data encrypted within the pTPM, and fully virtual objects. Furthermore, it should empower a vTPM user to attest to the protection level of these objects.

The first option requires no modifications to the pTPM interface. It involves the vTPM interface invoking the pTPM object creation command and fetching the public and private components of the object, with the private portion encrypted by the pTPM-protected parent key. When the vTPM user subsequently loads the object, the request is forwarded to the pTPM, and the vTPM returns the object handle to the user.

A vTPM software object refers to an object whose sensitive data can be accessed by the vTPM instance without or with minimal interaction with the pTPM. This feature makes our architecture suitable for applications relying on the vTPM for cryptographic operations rather than hardware-based security. Generally, software-based key management is expected to deliver superior performance compared to operations bound by the pTPM, which are constrained by the TPM access broker, resource management, and inherently limited computational resources. However, it's crucial to note that fully virtualized keys offer less protection compared to hardware-bound objects.

5.2.1 pTPMCreated attribute bit-mask

The pTPM does not need to understand the semantics of vTPM software objects, as they do not require interaction with the physical platform. Conversely, the vTPM interface should be aware of whether hardware-bound or software objects are being generated. This awareness can be achieved by introducing a new object attribute flag named *pTPMCreated*, which vTPM object creation functions can utilize to determine whether to invoke the pTPM or not. The TPM 2.0 specification [7] reserves several bits of the object attribute bit-mask for future additions, making any of them suitable for our purpose.

The *pTPMCreated* object attribute flag has the following implications:

- SET (1): The object's sensitive data is protected solely by the underlying pTPM;
- CLEAR (0): The object's sensitive data may exist in plaintext within the vTPM.

This attribute can also be applied to the VPS object, allowing the vTPM to be fully instantiated in software. While this approach may lower the security level of the instance, it facilitates a more flexible deployment of our vTPM architecture within a Cloud Service Provider infrastructure. In the case of hardware-bound VPS objects, they can be certified directly from the pTPM. It's important to note that the *pTPMCreated* object attribute is only relevant for a vTPM; pTPMs can either ignore it or mandate its use when supporting the proposed vTPM extension.

5.2.2 vTPM software object hierarchy

It is assumed that not all children of a vTPM software object need to be bound to a pTPM, which is why some may have *pTPMCreated* set to clear. This constraint simplifies the process of generating a vTPM software object. To create any object, the vTPM necessitates the parent key to be loaded beforehand. This becomes non-trivial when a vTPM object with *pTPMCreated* set is wrapped by a parent key that was not generated by the pTPM. As a result, a pTPM would only accept the *pTPMCreated* bit for the parent key to be set when it is invoked by the vTPM for object or key generation.

Generating a software object from a hardware-bound parent key, as depicted in Figure 5.2, presents research challenges. The first software object in each hierarchy has a parent with *pTPMCreated* set, requiring its parent to be loaded in the pTPM first. Therefore, the intermediary stage between hardware-protected keys and fully virtualized objects inevitably involves an interaction with the pTPM, as per the design.

5.2.3 Approaches

Considering the need for an intermediate phase as explained above, two possible approaches have been defined for this purpose: Software object created as unsealed pTPM SDO or Software object created as pTPM duplicable object.

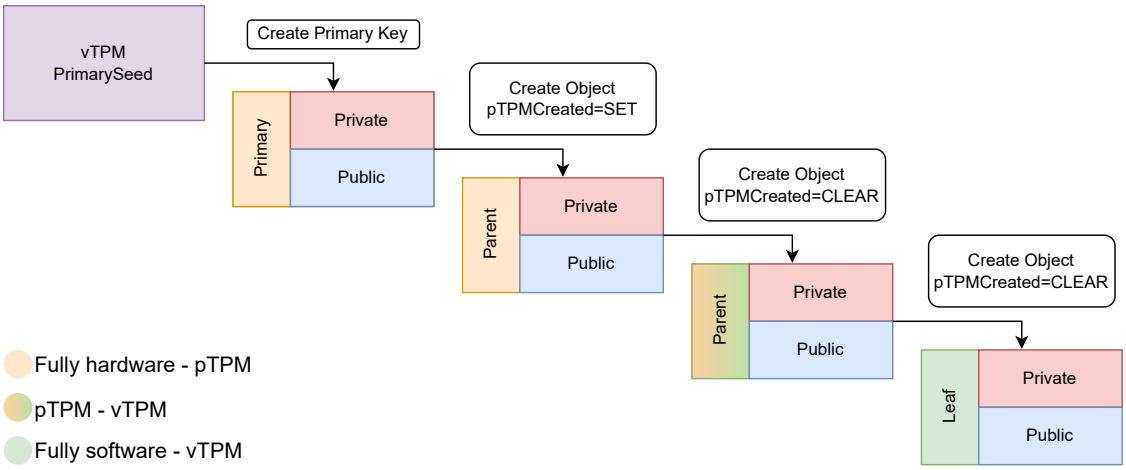

Figure 5.2. Fully softwarised object chain

Unsealed pTPM SDO

The TPM 2.0 command specification [23] mandates that the SDO must be generated within the context of a parent key. This ensures that its value, which is encrypted using a symmetric key, can later be exported without encryption. Consequently, one feasible approach for software objects with a parent protected by the physical TPM (pTPM) involves storing their sensitive data as payloads within pTPM SDOs, regardless of the specific object type requested through the vTPM interface.

To implement this approach, the vTPM software would need to convert the object creation process into a keyed hash object type and generate the sensitive data area within the software. This generated sensitive data area would then be presented to the pTPM as a standard SDO. When retrieved, the resulting private and public areas would be returned by the vTPM based on the original object template, rather than using the pTPM-generated SDO.

During the loading process, the vTPM would follow these steps:

1. Convert the object's public area template into an SDO format;
2. Load both the public and private areas into the pTPM, acquiring the SDO handle;
3. Access the sensitive data area of the object using its handle within the pTPM;
4. Store the sensitive data area of the object, along with the original object template, in the vTPM's memory.

It's important to highlight that this method presents significant challenges. There's a requisite for comprehensive mapping between the real object type and the SDO during both the creation and loading phases. Furthermore, the SDO's sensitive data area has a size constraint, pegged at 128 bytes as per the TPM 2.0 specification. Given this restriction, the vTPM might need to preserve components

vital for key regeneration instead of the entire TPM private object structure. This structure may overcome the size constraint of the SDO, for example the disk size of the RSA-2048 private object is 224 bytes.

pTPM duplicable object

Second approach involves utilizing the key duplication capabilities within TPM 2.0. The TPM allows for the movement of an object to another parent, provided that its attributes do not prohibit duplication, and its authorization policy explicitly permits it. The duplication process involves two encryption phases. The first phase applies inner wrapping using a symmetric key shared between the original and duplication platforms. The second phase, known as outer wrapping, encrypts the object using the algorithms of the new parent key. The outcome of this duplication process is the private area of the object, which is intended to be loaded by the new parent.

However, to put this process into action, it necessitates the creation of a vTPM software parent key in conjunction with the pTPM-protected hierarchies to enable the transfer of the software object. Additionally, if a new parent key is introduced, the software object can't be reloaded under the initial parent key. To circumvent this challenge, the strategy can employ a specialized key duplication method provided by the TPM. When executing the duplication command, the following conditions must be met:

- The object attributes fixedParent and encryptedDuplication are both clear;
- The duplication parent handle is set to the Null hierarchy handle;
- The symmetric encryption key size is 0, and its algorithm is set to a null value.

These parameters allow for duplicating the object in plaintext, bypassing both outer and inner wrappings, achieving the same result as the unseal operation. During the loading process, the vTPM would follow these steps:

1. Load the public and private areas into the pTPM, obtaining the object handle;
2. Duplicate the object to a null parent without any encryption, retrieving its sensitive area;
3. Store the sensitive area of the object, along with its template, in the vTPM's memory.

The advantage of this approach is its utilization of standard TPM structures and commands for object creation, eliminating the need to map the vTPM-requested object to a different pTPM type. Nonetheless, a potential drawback lies in managing the object authorization policy for duplication, as this command necessitates the vTPM to activate a policy session. Moreover, the vTPM should present a distinct authorization policy compared to the pTPM object, creating a policy mapping challenge within the vTPM.

5.3 Store and restore vTPM state

This section delves into the makeup of the vTPM's state and the safeguarding mechanisms that can be activated through the vTPM manager, all of which are aimed at securing the virtual instance's state against potential tampering when it undergoes stoppage or restoration at various points in its lifecycle.

5.3.1 vTPM State

The vTPM instance comprises a collection of handles that reference various objects within the vTPM. These handles have different characteristics depending on their relationship with the underlying pTPM:

- Hardware-Bound Reserved Handles: These handles are associated with persistent hierarchy structures and specify the Virtual Primary Seed (VPS) objects, along with their authorization and policies;
- pTPM-Created Object Handles: These handles correspond to primary keys and hardware-bound child objects generated by the pTPM;
- vTPM Software Object Handles: These handles are related to vTPM software objects;
- Other Internal vTPM Handles: These handles are utilized for the vTPM's internal functions.

In this design, the vTPM state incorporates a handle map that associates each entry, 32-bit number, with a specific data structure. Each entry corresponds to a vTPM virtual handle known as a vHandle, and its value is mapped to a pTPM object through its pTPM handle, referred to as a pHandle. Specifically:

- Persistent hierarchy handles should map to VPS handles in the pTPM;
- Objects generated by the pTPM should map to actual hardware handles, enabling reference when interacting with the physical device;
- vTPM software objects should be mapped to a null value, signifying that they are solely persisted within the vTPM state.

The vTPM's permanent state includes the following data, as illustrated in Figure 5.3:

- Public and private areas of the VPS objects, as provided by the pTPM;
- Handles of the SWK (Software Key) keys required for loading VPS objects;
- NVRAM content, including user-defined indexes and objects stored within the vTPM;
- The aforementioned handle map.

Due to the size of this dataset, it cannot be safeguarded by the pTPM as a SDO. Consequently, a primary encryption key will be used to perform state encryption.

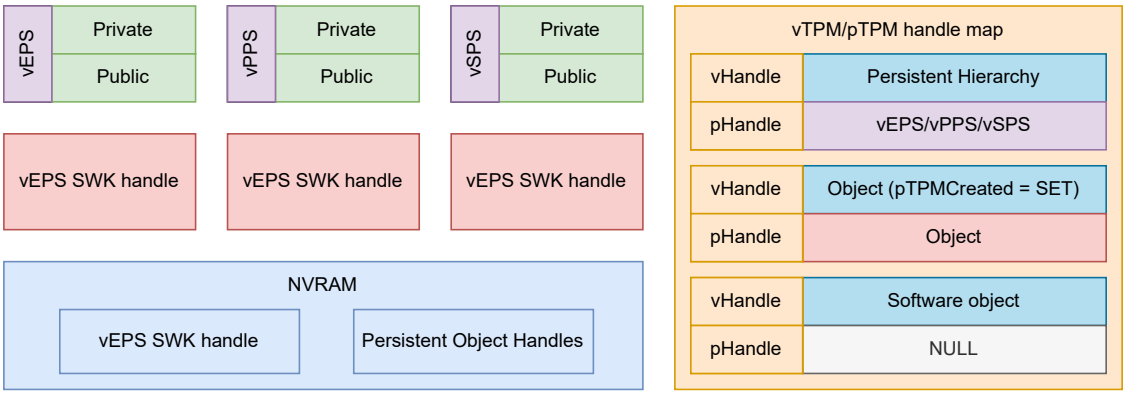

Figure 5.3. vTPM state

5.3.2 Storage and restoration

The fundamental procedure for storing the vTPM instance is integrated into the shutdown command, as outlined in the specification. Conversely, the state is restored as part of the init command, provided an encrypted state blob has been supplied by the vTPM manager during instantiation.

During vTPM shutdown, the data to be encrypted in the state does not contain the public and private areas of the primary seed. These primary seed areas are already secured through wrapping by the pTPM parent keys. The remaining state components, such as NVRAM content and the handle map, are encrypted by the pTPM using a symmetric algorithm. The resulting data blob is then presented to the vTPM manager. Essentially, the pTPM can be employed to encrypt arbitrary data through symmetric encryption when the instance is halted and decrypt it during restoration.

In the initialization phase of the vTPM, when an encrypted state blob is provided by the vTPM manager alongside the handles of vEPS, vPPS, and vSPS parent keys, the state is decrypted by the pTPM. The VSPK is regenerated by the vTPM instance under the vTPM Platform hierarchy, utilizing the known template. Subsequently, this key is utilized by the pTPM to decrypt the encrypted state blob. As the VSPK is a vTPM primary object, it is reconstructed from the vPPS and can't be permanently stored anywhere since the pTPM does not disclose private components. Once the state blob is decrypted, the vTPM proceeds to restore the persisted objects. Additionally, for each of the hardware-protected vTPM objects, the corresponding NVRAM Persistent Object index must be updated to align with the index in the pTPM NVRAM. This update is necessary because vTPM persistent objects are not genuinely stored in the pTPM once the virtual instance is stopped; thus, they need to be reloaded during vTPM state restoration.

Chapter 6

Implementation

In this chapter will be described the proof of concept implementation, highlighting the pTPM/vTPM commands that need extension and addition. They will be presented the processes for initialization, software object creation, as well as state stora and restore operation. Furthermore, it will be outlined the methods devised to strengthen the solution, accompanied by the relative modifications made to both the TSS code and the Microsoft TPM simulator software [24].

6.1 TSS and TPM simulator code basis

After presenting the components of the TSS and introducing the design of the solution it is important to describe how the source code of the TSS itself and of the Microsoft TPM simulator is structured to fully comprehend the design implementation.

6.1.1 TSS code structures and functions

ESAPI function structure

After a command setup, the standard procedure for utilizing TPM involves first invoking `Esys_<COMMAND_NAME>_Async` and then `Esys_<COMMAND_NAME>_Finish`. Internally, these execute the following operations:

1. Utilizes the System API for formatting, parsing, and managing send/receive operations with the TPM.
2. Generates outgoing nonces while processing incoming ones.
3. Encrypts the command parameters and decrypts the received response parameters.

In details:

- The Esys_<COMMAND_NAME>_Async template is designed to prompt the ESAPI to execute a TPM command asynchronously. This ensures that the operation doesn't halt while the TPM processes the command. Subsequent to this, Esys_<COMMAND_NAME>_Finish should be invoked to obtain the TPM's response parameters.
- The Esys_<COMMAND_NAME>_Finish template facilitates the ESAPI in retrieving the outcomes of an asynchronous TPM command. The operation's behavior is contingent on the timeout determined via Esys_SetTimeout, set to non-blocking by default. It's essential to first call Esys_<COMMAND_NAME>_Async before this function to activate the TPM command in an asynchronous mode. If the TPM provides a name for a specific object, the ESAPI records these parameters within the affiliated ESYS_TR object. Consequently, these details aren't relayed back to the user.

SAPI function structure

The SAPI function APIs are divided into various groups, among which are: command preparation, command execution, and command completion. The ESAPI commands introduced earlier internally use several SAPI commands, specifically:

- The Tss2_Sys_<COMMAND_NAME>_Prepare function offers the necessary information to applications, allowing them to compute a Command Parameters Hash (cpHash) for the command. This cpHash is used by the application in determining HMAC authorizations and pre-computing policy hashes, which are then employed when creating objects that utilize policy authorizations.
- The Tss2_Sys_ExecuteAsync function invokes the TCTI transmit callback to dispatch the TPM command stream. However, it doesn't trigger the receive function. While this function operates in a blocking manner, it's designed to yield a response promptly. It should only be invoked once following a Tss2_Sys_<COMMAND_NAME>_Prepare if the TPM command is successful. If the TPM command falters and Tss2_Sys_ExecuteFinish indicates a TPM error, this function can be reinvoked to resend the identical command buffer.
- The Esys_<COMMAND_NAME>_Finish function triggers the receive callback to obtain the response stream after invocation of Tss2_Sys_ExecuteAsync.
- The Tss2_Sys_<COMMAND_NAME>_Complete function extracts the response parameters and handles from a TPM command executed earlier. Should the caller not need a specific parameter or handle, they can input NULL for any of the response handles or parameters, and those values won't be returned. It's imperative to invoke this function only after Tss2_Sys_ExecuteAsync and prior to the subsequent Tss2_Sys_<COMMAND_NAME>_Prepare call.

Marshaling/Unmarshaling

Both the SAPI and ESAPI necessitate marshaling and unmarshaling, which consists of converting the memory representation of an object into a format apt for storage

or transmission. Instead of creating distinct marshaling and unmarshaling codes for the SAPI and ESAPI, this code has been consolidated into a shared namespace utilized by both.

6.1.2 TPM simulator code overview

In order to implement the new methods and the storage of some information internally to the vTPM there is the need to introduce some of the main components and characteristics of the simulator itself: vTPM server, the vendor specific flag and finally the variables defined as EXTERN.

TPM server

The simulator exposes its functionality, following an initial initialization phase, via socket using the `bool TpmServer(SOCKET s)` method which reads the raw bytes sent on the socket and triggers the execution of commands. Once the command operations have been performed, the method has the task of writing the response bytes onto the socket. A limitation of this is that the simulator can only respond to one command at a time.

Vendor specific flag

Since the proposed design involves the addition of new commands compared to those in the specifications, it is necessary to allow user-defined commands in the software. This possibility is created by a special flag inserted in the definition of the command itself, i.e. the *vendor - V* flag. However, it is important to remember that this addition enables the processing of the command once its execution has been triggered; therefore, it is necessary to also develop TSS functions that enable this trigger.

EXTERN variables

Last but not least, there is a need for the solution to store data to track status and binding with the pTPM. When examining the code and especially in the header file `Global.h`, you can notice that there are several variables defined as EXTERN and used to store volatile information such as handles or user-defined indexes. This is exactly where the aforementioned informations are stored, also as EXTERN variables.

6.2 Proof of Concept implementation

This section describes the implementation, including the interactions between virtual and physical TPM, of the proposed solution. All procedures will be described in terms of analyzed and developed code.

6.2.1 Extension functions

The functions to be extended internal to the pTPM will be presented below. These methods will then be used in the initialization procedures, store restore and creation of fully softwarised objects.

CreateSeed and LoadSeed

First two function are `TPM2_VIRT_CreateSeed` and `TPM2_VIRT_LoadSeed` which are a specialized versions of the standard `TPM2_Create` and `TPM2_Load` commands. As the name tells us they are used to create and load the seeds, i.e. VPSs, into the pTPM. The difference between `TPM2_VIRT_CreateSeed` and its standard counterparts is that before creating the requested objects, their attributes are validating checking their input public template. Following design specification both the sign and decrypt attributes have to be set to CLEAR and the restricted one to SET. After validation is completed the sensitive area of the objects (the seed itself) is populated triggering pTPM RNG function to generate a random number whose size has bees specified as input parameter by the vTPM. The `TPM2_VIRT_LoadSeed`, one again verifies that objects attributes are consistent and then loads public and private parts of the seed inside pTPM returning the so called VPS handle.

Virtual CreatePrimary

The `TPM2_VIRT_CreatePrimary` command corresponds to the virtual counterpart of `TPM2_CreatePrimary` command. They are so similar with the difference that the latter generates a primary key handle based on a hierarchy permanent handle, that identifies its hierarchy, and a public template, the first instead internally checks that the input handle's association with a loaded vTPM seed rather than a persistent hierarchy internal handle, by examining its object attribute. Furthermore, sensitive area of the vTPM seed is used as input for `DRBG_InstantiateSeeded` KDF in order to generate primary key using pTPM hardware.

StoreState and RestoreState

The commands to leverage on for state storing and restoring are `TPM2_VIRT_StoreState` and `TPM2_VIRT_RestoreState` which perform symmetric encryption and decryption similar to the standard `TPM2_EncryptDecrypt2` function. The key used for protecting the state is generated from the pTPM through the `TPM2_VIRT_CreatePrimary` function during the vTPM instantiation process. A limitation of these methods is the maximum data blob size which is limited to 1024 bytes, for this reasons the state buffer is divided in chunks of that size and then reassembled together.

6.2.2 Initialisation procedure

When binding with pTPM is required, the vTPM cannot be created and used unless following a precise procedure detailed below. This involves three main actors: vTPM Manager, pTPM and vTPM.

Command mappings and interactions

The vTPM manager begins the binding procedure and has the task of contacting the pTPM for the creation of the SWKs and then making them available to the various vTPMs who will subsequently use them for the creation of the seeds and the state protection key.

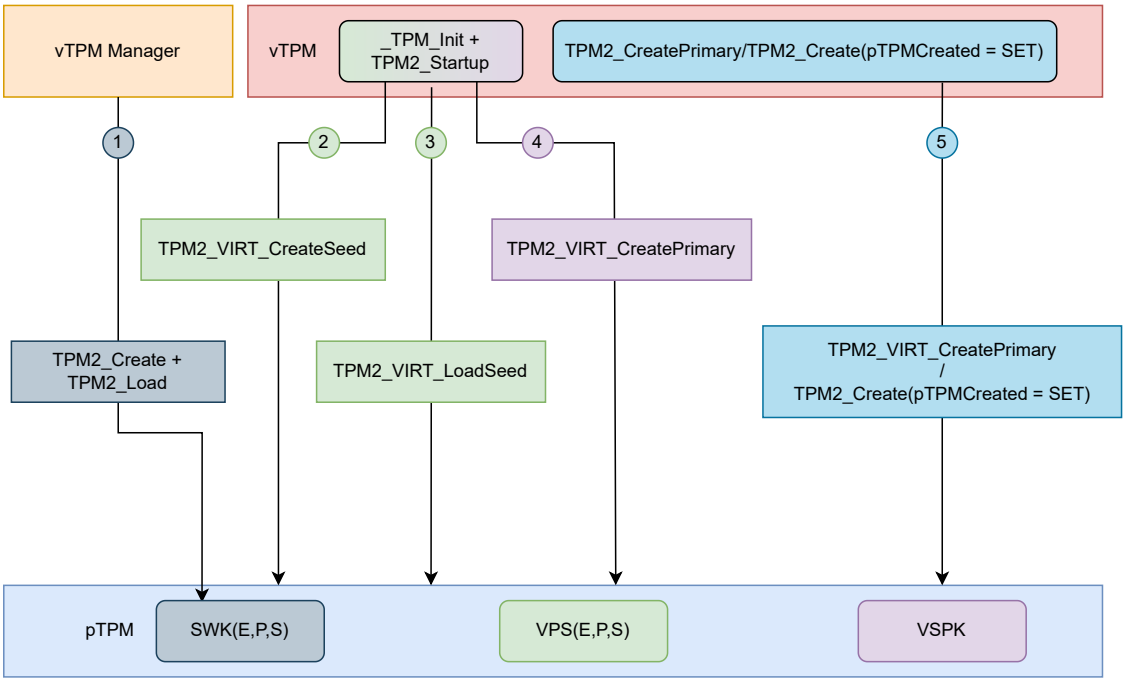

Figure 6.1. Command mappings for initialisation procedure

The Figure 6.1 outlines the different steps involved in this procedure:

1. The vTPM manager communicates with the pTPM interface to generate and load SWKs, one key for each persistent hierarchy. This is achieved by issuing `TPM2_Create` and `TPM2_Load` command saving their handle in a dedicated file shared among vTPM instances. Moreover it prepares a VSPK public template for each vTPM instance that have to be created;

2. The vTPM instances begin binding with pTPM issuing `TPM2_VIRT_CreateSeed` for each persistend hierarchy, using the provided SWK handles;

3. The vTPM instance queries the pTPM using the `TPM2_VIRT_LoadSeed` command for each VPS object, storing the corresponding pVPSHandle in its handle map;

4. The vTPM instance contact the pTPM for the creation of the VSPK triggering `TPM2_VIRT_CreatePrimary` command and stores the pVSPKHandle in the handle map, associated with an internal `VPSK_STRUCTURE` object that is not part of the standard TPM 2.0 handles.

5. Once previous steps have been successfully completed and subsequently, after sending the startup command to the vTPM, it will be available for use. At this moment the vTPM user can generate its own key using the standard `TPM2_Create` and `TPM2_CreatePrimary`, that in turn will be forwarded to pTPM if hardware bind is required, i.e. *pTPMCreated* SET.

This implementation differs from the one presented in the [2] as the vTPMs are not created automatically by the vTPM manager, leaving the creation procedure to the user by running the appropriately modified simulator binary.

vTPM Manager

The vTPM manager in this implementation is defined as a class in C++. Its constructor creates the paths where the different configuration files will be hosted such as the file containing the SWKs and the specific files for each vTPM which in turn contain a file for the VSPK template and a file where the state will be saved during the storestate phase detailed below. Its methods `CreateSWK(ESYS_TR)` and `DefineVSPKPublicTemplate()` are the ones that perform the above operations, the first creates unique keys for all vTPMs and the second will create a VSPK template for each vTPM.

vTPM init and binding

Internally to vTPM binding operations are part of the `_TPM_Init` method. A set of utility methods has been defined and implemented to be able to make a connection to the pTPM and trigger the execution of commands.

6.2.3 Fully Softwarised Object

As explained in the previous chapter, to provide greater flexibility in a cloud environment it is necessary for the user to have the ability to create objects entirely in software without having any tight connection to the hardware. This section details the procedure for creating the first object not linked to the pTPM.

Command mappings and interactions

As depicted in Figure 6.2, the procedure comprises several steps:

1. The vTPM user initiates a `TPM2_Create` command with the *pTPMCreated* bit set to clear, providing the vObjectHandle parent handle. This vObjectHandle is subsequently used as handle for a `TPM2_Create` command within the pTPM. The pTPM utilizes the object handle pObjectHandle to create the object, successfully loading the private component of its parent object. The pTPM then returns both the private and public parts of the newly created object. This object may appear as either a SDO or a duplicable object that retains its original type based on the strategy chosen.

2. The vTPM user issues a `TPM2_Load` command through the vTPM interface, which is forwarded to the pTPM, loading the newly created software object into the pTPM interface;
3. The vTPM instance queries the pTPM using the `TPM2_VIRT_LoadSeed` command for each VPS object, storing the corresponding pVPSHandle in its handle map;
4. Depending on the strategy followed for the conversion to software, the vTPM then sends one of the following commands to the pTPM interface:
   - `TPM_Unseal`, if the software object is mapped to a standard SDO by the pTPM interface.
   - `TPM_Duplicate`, if the software object is duplicable without encryption and has a null parent.

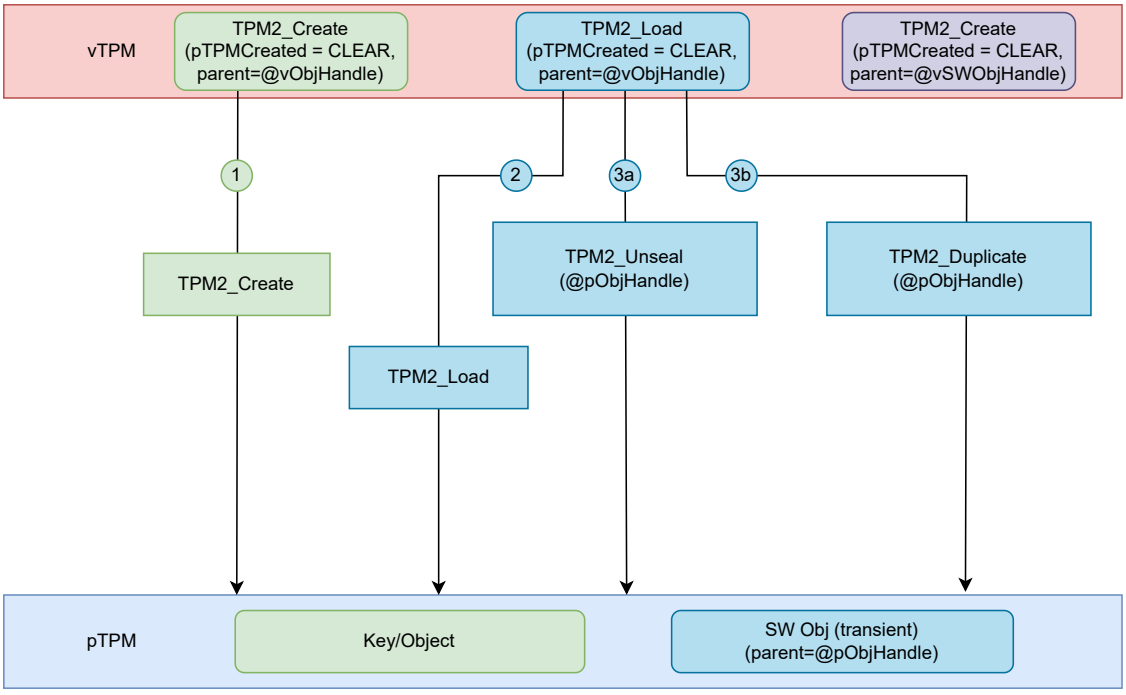

Figure 6.2. Command mappings for creation of first software object

In both scenarios, the sensitive portion of the object is returned to the vTPM interface, where it is stored in memory. After successful loading, the software object can be removed from the pTPM memory. Subsequent vTPM commands related to this object do not rely on pTPM commands because the object's vSWObjectHandle is mapped to a NULL value in the handle map. This signifies that the vTPM itself holds the sensitive portion of the software object.

6.2.4 Store and restore state

When shutting down the vTPM, the command `TPM2_VIRT_StoreState` is utilized to encrypt the vTPM's state and subsequently store it in a file. When the vTPM has

to be recreated, the state has to be restored and for this reason the `_TPM_Init` command internally performs the operation to address this issuing `TPM2_VIRT_RestoreState` command.

Command mappings and interactions

Figure 6.3 presents the commands and interactions necessary for shutting down and creating a vTPM with a previously saved state.

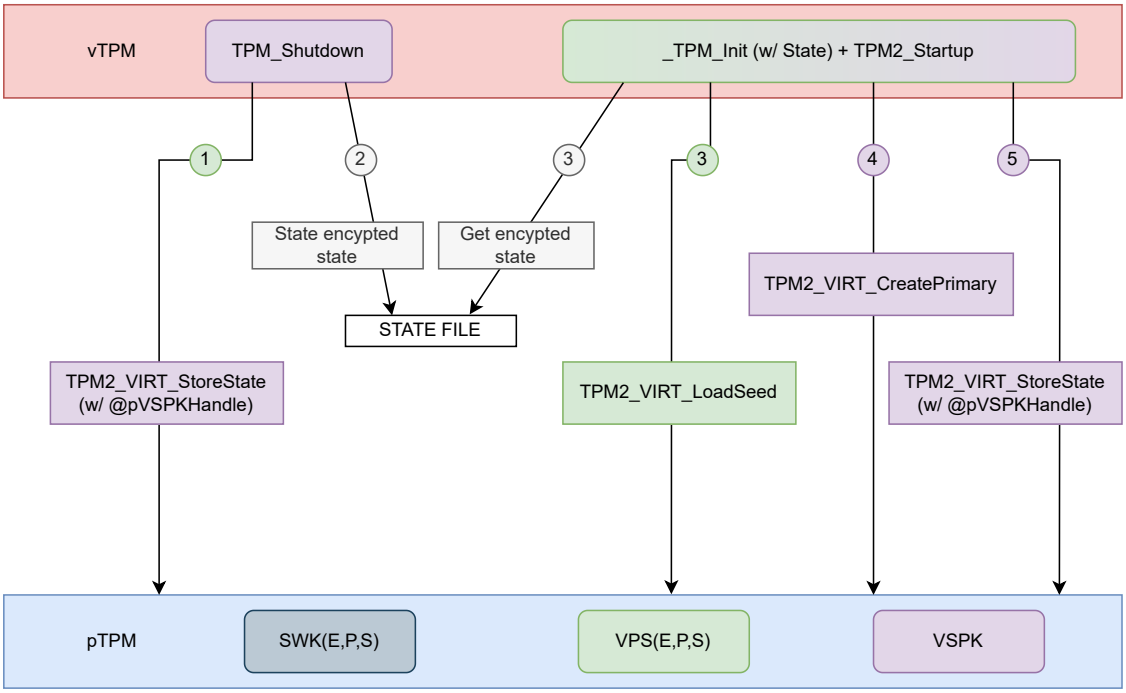

Figure 6.3. Command mappings for store and restore state operations

The order of commands to be issued is the following:

1. Upon receiving a `_TPM_Shutdown` command, the vTPM initiates a `TPM2_VIRT_StoreState` command directed at the pTPM. This command encrypts the vTPM's binary state representation using the VSPK;

2. The vTPM then saves this encrypted state, along with information needed for future restoration (such as the private and public parts of the VPS), into a file;

3. At a later stage, when the vTPM instance has to be restored, `_TPM_Init` command internally triggers a `TPM2_VIRT_LoadSeed` command for each VPS object, then avoiding the recreation of primary seeds at each restoration;

4. The vTPM instance issues a `TPM2_VIRT_CreatePrimary` command to the pTPM, utilizing the VSPK template to regenerate the exact same key. The resulting pVSPKHandle is then restored in the handle map;

5. Finally, the vTPM instance executes a `TPM2_VIRT_RestoreState` command against pTPM. This command uses the encrypted state, recovered from the previously saved file, and the VSPK handle for the decryption of the encrypted state itself.

State marshal/unmarshal phase and state encrypt/decrypt phase

To save and consequently encrypt the state, it was essential to perform a marshaling operation on the data structures within the vTPM State. Due to the restricted size of the data blob provided as input to the storage function, akin to the encrypt-decrypt process, it became imperative to segment the data blob into separate chunks, encrypt each of them individually, and subsequently reassemble them before storing them in the file, as previously elucidated.

Likewise, during the restoration phase, the encrypted data blob needed to be initially divided into segments, followed by decryption, and then the concluding unmarshaling operation.

6.3 Enabling hardware binding

As the proposed solution entails modifications to the pTPM code, we've incorporated an instance of the simulator for simulating the updated pTPM. To ensure the simulator can be initiated in both physical and virtual modes, adjustments have been applied to the input parameters.

6.3.1 Simulator input parameters necessary for binding

The unaltered Microsoft simulator offers the possibility to use the `-m (--manufacture)` option to force the re-manufacturing of the NV state of the TPM and to specify a port number for receiving commands. In addition to these commands, the new version introduces two additional options:

- The `-b (--hwbind)` used to indicate whether a pTPM binding is required;
- The `-r (--restore)` used to indicate whether the vTPM to be created has to restore its state.

If the first option has been set, it will be necessary to provide three paths in the command: the first to indicate where the file containing the SWKs is saved; the second to indicate where the VSPK public template is present and the third to indicate the file where to save/restore the state.

Chapter 7

Conclusions and Future Work

This chapter will describe the results and conclusions drawn from research questions and objectives of the thesis work. The weak points will be analyzed providing an idea on possible future implementations aimed at resolving the latter acting as a guide and directing future researchers.

In conclusion, the proof of concept presented in this work adheres to the requirements stipulated by the solution design. The goal of establishing a robust link between the virtual and physical components was successfully achieved, seamlessly obscuring the underlying procedures from the end user and thereby preserving the existing security flows that leverage this technology. Additionally, the capability to create entirely software-based objects was ensured through the processes that facilitate the transition from a fully hardware-based approach to a completely software-oriented one. In closing, it is pertinent to highlight the feasibility of restoring and/or migrating the vTPM under an alternative pTPM. This provides flexibility to Cloud Service Providers, who often necessitate migrating virtual machines, and consequently the vTPM, to different hardware for resource management purposes.

One notable drawback of this implementation resides in its entirely software-based nature. Given the non possible modification of the internal procedures of the commercially available pTPMs, a decision was made to pursue a wholly software-centric solution. Consequently, this approach serves exclusively to demonstrate actual feasibility and practicality, rather than providing a robust solution. The simulator employed in this work generates an instance of vTPM that is not supported by any hypervisor, as it serves merely as a reference implementation. As such, the capability to execute operations, such as Remote Attestation, is not within its purview. This limitation underscores a critical challenge in translating the theoretical and simulated outcomes into practical, real-world applications.

Regarding future developments, it is prudent to commence with the last limitation highlighted, specifically, the restriction concerning the solution's application with a hypervisor. An initial exploration of the source code and documentation of IBM's vTPM [25] and supported by QEMU, reveals that it is grounded in the reference implementation of Microsoft's TPM 2.0, which was utilized in this work. Consequently, a viable avenue for future development may encompass the implementation of the QEMU frontend and backend driver, thereby facilitating communication between the vTPM, in Guest VM, and its physical counterpart, in the

Host VM.

Following this proposal it is imperative to safeguard vTPM memory within a secure and isolated environment, such as an SGX enclave. Given that the version of the simulator developed by IBM also maps the non-volatile memory into a file, it too presents a vulnerability and lacks protection. Therefore, a solution wherein the vTPM Manager also oversees the encryption and decryption of the NVRAM could present an interesting pathway for future exploration and feedback. This approach would not only enhance the security of the vTPM memory but also fortify the robustness of the entire system against potential threats and vulnerabilities, ensuring that the integrity and confidentiality of the data are preserved.

Appendix A

Installation guide

As previously presented, the thesis project is based on the Microsoft TPM2 Simulator and TSS2 Software github project, these projects have been extended and the new software is contained in the sources attached to the thesis itself offering the possibility of creating pTPM and vTPM in software.

In order to replicate a scenario in which a vTPM carries out the binding procedure with the pTPM it is necessary to install the following components:

- **tpm2-tss**: hosts source code implementing the Trusted Computing Group's (TCG) TPM2 Software Stack (TSS);
- **ms-tpm-20-ref**: official TCG reference implementation of the TPM 2.0 Specification.

The following guide was tested under Linux Ubuntu operating system version $\geq$ 20.04

A.1 tpm2-tss

To install this library, you must first install its dependencies and then proceed to install the library from sources.

Installing dependencies

The dependencies required for tpm2-tss are the following:

- GNU Autoconf
- GNU Autoconf Archive, version $\geq$ 2019.01.06
- GNU Automake
- GNU Libtool

- C compiler
- C library development libraries and header files
- pkg-config
- doxygen
- OpenSSL development libraries and header files, version $\geq$ 1.1.0
- libcurl development libraries
- Access Control List utility (acl)
- JSON C Development library
- Package libusb-1.0-0-dev

To install them run the following commands in the terminal:

```
$ sudo apt -y update
$ sudo apt -y install \
  autoconf-archive \
  libcmocka0 \
  libcmocka-dev \
  procps \
  iproute2 \
  build-essential \
  git \
  pkg-config \
  gcc \
  libtool \
  automake \
  libssl-dev \
  uthash-dev \
  autoconf \
  doxygen \
  libjson-c-dev \
  libini-config-dev \
  libcurl4-openssl-dev \
  uuid-dev \
  libltdl-dev \
  libusb-1.0-0-dev \
  libftdi-dev
```

Bootstrapping the Build

To configure the tpm2-tss source code first run the bootstrap script, which generates list of source files, and creates the configure script:

```
$ ./bootstrap
```

Configuring the Build

Then run the configure script, which generates the makefiles:

```
$ ./configure--with-udevrulesdir=/etc/udev/rules.d
   --with-udevrulesprefix
```

Compiling the Libraries

Then compile the code using make:

```
$ make -j$(nproc)
```

Installing the Libraries

Once you've built the tpm2-tss software it can be installed with:

```
$ sudo make install
```

Post-installation steps

After installing the udev rule in the appropriate directory for your distribution, it's necessary to prompt udev to reload its rules and implement the new one. Generally, this can be executed with the command below:

```
$ sudo udevadm control --reload-rules && sudo udevadm trigger
```

It may be necessary to run ldconfig (as root) to update the run-time bindings before executing a program that links against libsapi or a TCTI library:

```
$ sudo ldconfig
```

A.2 ms-tpm-20-ref

This extended implementation can be directly used via the TPM 2.0 simulator that emulates a TPM 2.0 device and can be accessed via a custom TCP based protocol. The installation is quite since it is only necessary to build, configure and then install the project:

```
$ cd ./TPMCmd && \
./bootstrap && \
./configure && \
sudo make && \
sudo make install
```

It is possible to enable standard DEBUG, the one implemented by Microsoft, and the custom DEBUG implemented for the purpose of this thesis. Custom DEBUG will build a TPM simulator which is verbose during binding procedure, to enable this it is necessary to add `VARS=-DMYDEBUG` at the make command.

Please note that, currently, the TPM simulator does not support OpenSSL version 3, in order to success the installation you need to:

1. remove newer OpenSSL version:

   ```
   $ sudo apt remove openssl
   ```

2. download and install older version, i.e. 1.1.1w:

   ```
   $ wget https://www.openssl.org/source/openssl-1.1.1w.tar.gz
      && tar -xvzf openssl-1.1.1w.tar.gz && cd openssl-1.1.1w
      && ./config && make && make install
   ```

Appendix B

User Manual

The Proof of Concept developed in this thesis can be simply reproduced using materials attached to thesis itself, it is essential to install: TPM 2.0 extended Simulator and the extended tpm2-tss.

In particular the components that will be used are the following ones:

- pTPM/vTPM: instantiated as TPM 2.0 simulator, they will simulate the physical and virtual chip and will be the main actors in the binding procedure;
- vTPM Managar: script that acts as initiator of procedure, it will prepare the environment to perform a successful binding;
- VM (optional): it is possible to create a VM and attach it to the default network, that where pTPM and vTPM will be created.

Testing

First create a folder that will contain NVRAM of the pTPM and then create it running:

```
$ tpm2-simulator -m
```

It should print on terminal as follows:

```
LIBRARY_COMPATIBILITY_CHECK is ON
Manufacturing NV state...
Size of OBJECT = 1204
Size of components in MSSIMT_SENSITIVE = 744
    MSSIMI_ALG_PUBLIC 2
    MSSIM2B_AUTH 50
    MSSIM2B_DIGEST 50
    MSSIMU_SENSITIVE_COMPOSITE 642
MAX_CONTEXT_SIZE can be reduced to 1264 (1344)
TPM command server listening on port 2321
Platform server listening on port 2322
```

Once created the pTPM it is possible to run vTPM Manager building and running the script, from main folder run:

```
$ make && ./build/run_test
```

Script will request to prompt:

1. Number of vTPM instaces (max 3) for which setup the environment, i.e. VSPK public template;
2. Path where SWKs will be stored;
3. Path where vTPM parameters will be stored, note that this will be requested many times as the number of vTPM.

Depending if DEBUG mode is enabled, passing `VARS=-DMYDEBUG` to make command, the script run in interactive and verbose mode or not. Without DEBUG mode the expected output is:

```
*** vTPMManager ***
How many vTPM do you want to instantiate? (MAX 3)
1
Provides me path to store swk context
/path_to_swk
Provides me path to folder to store vTPM#1 parameters
/path_to_vTPM

*** vTPMManager Startup ***

*** Creation of the ENDORSEMENT SWK (eSWK) ***

*** Creation of the OWNER SWK (sSWK) ***

*** Creation of the PLATFORM SWK (pSWK) ***

*** Definition of VSPK Public Template for each vTPM instance ***
```

Now it is possible to proceed with the last phase, i.e. the one in which the vTPM is instantiated by performing the binding operations. Create a new folder as for the pTPM but for the vTPM and from inside run:

```
$ tpm2-simulator -m -b port_number /path_to_swk
   /path_to_vTPM/vspk.dat /path_to_vTPM/state.dat
```

Where the paths are the same indicated earlier during vTPM Manager execution and port is to be chosen, note that the ports 2321 and 2322 are used by pTPM which has this constraints. Once again the output differs if TPM simulator has been built with verbosity enabled or not. In case it is not enabled the output is silent and is the same of the pTPM with the only difference on the exposed port.

Bibliography

[1] ETSI, "Network Functions Virtualisation (NFV); Trust; Report on Attestation Technologies and Practices for Secure Deployments", DGR/NFV-SEC007, October 2017, https://www.etsi.org/deliver/etsi_gr/NFV-SEC/001_099/007/01.01.01_60/gr_nfv-sec007v010101p.pdf

[2] M. De Benedictis, L. Jacquin, I. Pedone, A. Atzeni, and A. Lioy, "A novel architecture to virtualise a hardware-bound trusted platform module", Future Generation Computer Systems, vol. 150, January 2024, pp. 21–36, DOI 10.1016/j.future.2023.08.012

[3] S. Berger, R. Caceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vTPM: Virtualizing the trusted platform module", 15th USENIX Security Symposium (USENIX Security 06), Vancouver (Canada), July 31-August 4, 2006, pp. 305–320

[4] X. Wan, Z. Xiao, and Y. Ren, "Building trust into cloud computing using virtualization of TPM", 2012 Fourth International Conference on Multimedia Information Networking and Security, Nanjing (China), November 2-4, 2012, pp. 59–63, DOI 10.1109/MINES.2012.82

[5] G. Arfaoui, P.-A. Fouque, T. Jacques, P. Lafourcade, A. Nedelcu, C. Onete, and L. Robert, "A cryptographic view of deep-attestation, or how to do provably-secure layer-linking", Applied Cryptography and Network Security, Rome (Italy), June 20-23, 2022, pp. 399–418, DOI 10.1007/978-3-031-09234-3_20

[6] J. Wang, F. Xiao, J. Huang, D. Zha, C. Fan, W. Hu, and H. Zhang, "A security-enhanced vTPM 2.0 for cloud computing", Information and Communications Security, Beijing (China), December 6-8, 2017, pp. 557–566, DOI 10.1007/978-3-319-89500-0_48

[7] Trusted Computing Group, "Trusted Platform Module Library Part 2: Structures", November 2019, https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part2_Structures_pub.pdf

[8] Trusted Computing Group, "TCG TSS 2.0 Overview and Common Structures Specification", September 2021, https://trustedcomputinggroup.org/wp-content/uploads/TSS_Overview_Common_v1_r10_pub09232021.pdf

[9] Trusted Computing Group, "TCG Feature API (FAPI) Specification", June 2020, https://trustedcomputinggroup.org/wp-content/uploads/TSS_FAPI_v0p94_r09_pub.pdf

[10] Trusted Computing Group, "TCG TSS 2.0 Enhanced System API (ESAPI) Specification", October 2021, https://trustedcomputinggroup.org/wp-content/uploads/TSS_ESAPI_v1p0_r14_pub10012021.pdf

[11] Trusted Computing Group, "TCG TSS 2.0 System Level API (SAPI) Specification", October 2021, https://trustedcomputinggroup.org/wp-content/uploads/TSS_SAPI_v1p1_r36_pub10012021.pdf

[12] Trusted Computing Group, "TCG TSS 2.0 TPM Command Transmission Interface (TCTI) API Specification", January 2020, https://trustedcomputinggroup.org/wp-content/uploads/TCG_TSS_TCTI_v1p0_r18_pub.pdf

[13] Trusted Computing Group, "TCG TSS 2.0 TAB and Resource Manager Specification", April 2019, https://trustedcomputinggroup.org/wp-content/uploads/TSS_2p0_TAB_ResourceManager_v1p0_r18_04082019_pub.pdf

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

[15] NIST, "The NIST Definition of Cloud Computing", NIST SP800-145, September 2011, DOI 10.6028/NIST.SP.800-145

[16] C. S. Alliance, "Top Threats to Cloud Computing Pandemic Eleven", June 2022, https://cloudsecurityalliance.org/artifacts/top-threats-to-cloud-computing-pandemic-eleven

[17] P. Jauernig, A. Sadeghi, and E. Stapf, "Trusted execution environments: Properties, applications, and challenges", IEEE Security & Privacy, vol. 18, March-April 2020, pp. 56–60, DOI 10.1109/MSEC.2019.2947124

[18] F. X. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and V. R. Savagaonkar, "Innovative instructions and software model for isolated execution", Hardware and Architectural Support for Security and Privacy, Tel-Aviv (Israel), June 23-24, 2013, pp. 73–80, DOI 10.1145/2487726.2488368

[19] AMD Secure Encrypted Virtualization (SEV), https://www.amd.com/en/developer/sev.html

[20] P. Cheng, W. Ozga, E. Valdez, S. Ahmed, Z. Gu, H. Jamjoom, H. Franke, and J. Bottomley, "Intel TDX demystified: A top-down approach", 2303.15540v1, March 2023, DOI 10.48550/arXiv.2303.15540

[21] Trusted Computing Group, "Virtualized Trusted Platform Architecture Specification", September 2011, https://trustedcomputinggroup.org/wp-content/uploads/TCG_VPWG_Architecture_V1-0_R0-26_FINAL.pdf

[22] P. England and J. Loeser, "Para-virtualized TPM sharing", Trusted Computing - Challenges and Applications, Villach (Austria), March 11-12, 2008, pp. 119–132, DOI 10.1007/978-3-540-68979-9_9

[23] Trusted Computing Group, "Trusted Platform Module Library Part 3: Commands", November 2019, https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part3_Commands_pub.pdf

[24] Official TPM 2.0 Reference Implementation (by Microsoft), https://github.com/microsoft/ms-tpm-20-ref

[25] SWTPM - Software TPM Emulator, https://github.com/stefanberger/swtpm