A novel architecture to virtualise a hardware-bound trusted platform module

Future Generation Computer Systems 150 (2024) 21–36

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A novel architecture to virtualise a hardware-bound trusted platform module

Marco De Benedictis [a], Ludovic Jacquin [b], Ignazio Pedone [a,*], Andrea Atzeni [a], Antonio Lioy [a]

[a] Politecnico di Torino, Dip. Automatica e Informatica, Corso Duca Degli Abruzzi, 24, Torino, 10129, Italy
[b] Hewlett Packard Labs, Hewlett Packard Enterprise, Bristol, BS34 8QZ, United Kingdom

ARTICLE INFO

Article history:
Received 9 July 2022
Received in revised form 6 August 2023
Accepted 9 August 2023
Available online 12 August 2023

Keywords:
Computer security
Trust management
Platform virtualization
Cloud computing security
Trusted computing
Trusted platform module

ABSTRACT

Security and trust are particularly relevant in modern softwarised infrastructures, such as cloud environments, as applications are deployed on platforms owned by third parties, are publicly accessible on the Internet and can share the hardware with other tenants. Traditionally, operating systems and applications have leveraged hardware tamper-proof chips, such as the *Trusted Platform Modules* (TPMs) to implement security workflows, such as remote attestation, and to protect sensitive data against software attacks. This approach does not easily translate to the cloud environment, wherein the isolation provided by the hypervisor makes it impractical to leverage the hardware root of trust in the virtual domains. Moreover, the scalability needs of the cloud often collide with the scarce hardware resources and inherent limitations of TPMs. For this reason, existing implementations of *virtual TPMs* (vTPMs) are based on TPM emulators. Although more flexible and scalable, this approach is less secure. In fact, each vTPM is vulnerable to software attacks both at the virtualised and hypervisor levels. In this work, we propose a novel design for vTPMs that provides a binding to an underlying physical TPM; the new design, akin to a virtualisation extension for TPMs, extends the latest TPM 2.0 specification. We minimise the number of required additions to the TPM data structures and commands so that they do not require a new, non-backwards compatible version of the specification. Moreover, we support migration of vTPMs among TPM-equipped hosts, as this is considered a key feature in a highly virtualised environment. Finally, we propose a flexible approach to vTPM object creation that protects vTPM secrets either in hardware or software, depending on the required level of assurance.

1. Introduction

Modern cloud infrastructures require sophisticated security mechanisms, given the need for isolation and resource segregation of instances shared among different tenants. Within cloud applications, security-sensitive software can benefit from ready-to-use security workflows to manage cryptographic data and ensure the proper functioning of the environment, such as the ones proposed by the *Trusted Computing* (TC) paradigm.

TC, as defined by the *Trusted Computing Group* (TCG) [1], proposes mechanisms that can help secure an environment based on a hardware *Root of Trust* (RoT), i.e. the *Trusted Platform Module* (TPM). Two of those mechanisms are *Remote Attestation* (RA), to remotely assess the trustworthiness of a platform at a given time against a white-list of known-good values, and *sealing* of data via hardware-protected keys. Although widely used in commodity hardware platforms (e.g. by Microsoft Windows BitLocker [2]), TPM-based protection does not scale well in highly virtualised environments, such as clouds. This is due to the architectural limitations of the TPM, whose hardware resources and internal structures do not accommodate for multi-tenant execution environments. Because of this, it is impossible to share a single TPM among different *Virtual Machines* (VMs) or containers, although access to the physical device could be provided via specific hypervisor-level mechanisms, such as pass-through drivers.

Given these limitations, the research community proposed architectures to virtualise the TPM by means of a software implementation of the device, known as vTPM, and to expose this software to virtual instances. These architectures allow the porting of secure workflows that leverage a TPM in the virtualised environment in a seamless way, as the vTPM is available to the VM or container as a standard device.

However, as a software instance, protection of the vTPM resources (e.g. cryptographic keys) against software threats must be taken into account, e.g. to counter in-memory or side-channel attacks that may expose the state of the device to an adversary.

* Corresponding author.
E-mail address: ignazio.pedone@polito.it (I. Pedone).

https://doi.org/10.1016/j.future.2023.08.012

In this regard, research has addressed the need to establish a physical binding between one or more vTPM instances and the underlying *physical TPM* (pTPM), if available. This is provided by binding a vTPM identity credential, i.e. a vTPM identity certificate, to a pTPM. However, this approach does not allow to protect generic vTPM secrets or cryptographic material in the pTPM, which lowers the security level of the vTPM to a software process rather than a tamper-resistant device.

Other research proposals aim to protect the software vTPM via hypervisor-level security controls, or trusted execution environment, to enforce access control and strong integrity assurance.

However, to the best of our knowledge, the existing proposals do not address the entire vTPMs state binding with a pTPM. Furthermore, a secure and complete binding between vTPM and pTPM poses relevant flexibility issues, which our solution addresses, allowing for the migration of a virtual instance from one physical host to another. Thus, our solution makes the management of keys and sealed data practical in a vTPM, while using the underlying hardware-level protection. This solution also enables attestation of the vTPM-pTPM binding attestation, to enable a vTPM user to verify that its secrets are correctly protected by a pTPM.

1.1. Contribution

In the following, we highlighted the contribution of this work:

1. investigation of the data structures and commands of the TPM in its latest specification version, TPM 2.0, and selection of the vTPM elements that the pTPM shall protect to achieve hardware-level security. In this regard, we aim at minimising the number of elements protected by the pTPM to effectively provide an identity binding between the virtualised and the physical platform.
2. proposal of an extension of the existing TPM 2.0 architecture that embeds the required data structures and commands to support the hardware-bound vTPM management. In such a design, we aim to not modify existing TPM architectural elements (e.g. data structures, commands) so that our novel architecture does not collide with the existing specification. In fact, we extend the TPM 2.0 architecture with a limited set of additional structures and commands that do not overlap with the standards but are built from the existing components to maximise the affinity with the TCG specifications. Moreover, we do not exceed the existing constraints on hardware resources (e.g. buffer sizes) as described in the specifications. Our main goal is to minimise the number of requested changes to a pTPM architecture so that our extension is backwards compatible and can be deployed in exiting pTPM with a TPM firmware upgrade.
3. support for the migration of vTPMs among TPM 2.0 equipped hosts, and increased scalability of vTPMs in a highly virtualised environment (such as a private cloud) via a flexible hardware–software interaction between each vTPM and the underlying pTPM.
4. support for high-performance applications since our vTPM design allows for keys and other TPM objects to exist in software rather than in the pTPM. While this is possible, the vTPM user would have to request this specific behaviour. This should be enabled with care since might enables a vTPM to be fully softwarised (i.e. there is no interaction with a pTPM). Our proposed design enables a user to verify that a hardware-bound vTPM is correctly enforced by a pTPM when the user knows a trust anchor (e.g. a public part) of a pTPM-protected attestation key.

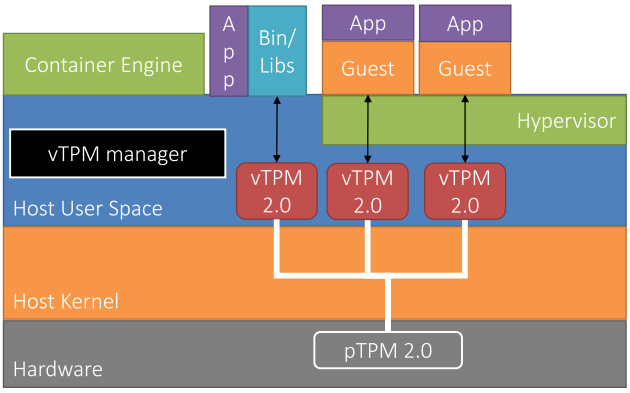

Fig. 1. vTPM target deployment.

1.2. Use case: Cloud provisioning of hardware-bound vTPM

A domain that can benefit from our proposal is the cloud environment, with particular attention to a *Infrastructure-as-a-Service* (IaaS) service model. This enables the sharing of physical resources (e.g. compute, network, storage) among several virtualised instances, i.e. VMs or containers, wherein cloud users can deploy their applications. The *Cloud Service Provider* (CSP) is the owner of the IaaS infrastructure and it typically proposes a portfolio of technical solutions wherein features and resources are selectable depending on the requirements of customer services and their subscription plan.

In this scenario, a CSP could offer specialised virtual instances that can leverage the TPM for the generation of cryptographic keys and secure storage of secrets, such as *Application Programming Interface* (API) keys. Moreover, the CSP itself could benefit from vTPM-equipped instances to implement security workflows, such as infrastructure attestation and detection of attacks on virtual tenants. Our proposal covers these use cases by enabling the instantiation of vTPMs in a hypervisor domain so that they can be presented as standard devices to VMs and containers, with enforcement by the hardware. Compared to existing approaches, our proposal retains the hardware-level protection of TPM sensitive information against software manipulations. From the user's perspective, the data managed by the vTPM should not be accessible by other cloud users or by the hypervisor itself. Our approach does not require the cloud user to trust the hypervisor, as the vTPM protected information are not stored by the CSP at the hypervisor level: the pTPM itself is used to protect such information so that it cannot be accessed by any user, including the hypervisor, without explicit authorisation by the vTPM owner. Moreover, a CSP could require the vTPM-equipped instance to migrate to a different host. Given that migration is a critical feature of highly virtualised environments, this feature should be supported by the vTPM itself. In our proposed scenario, we support migration among computing hosts equipped with TPM 2.0. Otherwise, existing security services would be disrupted, as all the secrets protected by a specific vTPM would not be recoverable by other vTPMs or even by the host system.

Fig. 1 depicts the envisioned target deployment for a vTPM architecture tailored for a heterogeneous virtualised environment. Each vTPM is deployed in the host user space thanks to a *vTPM manager*, whose role is to interact with either the container runtime or a hypervisor so that vTPMs can be spawned and exposed to virtual instances. Moreover, the vTPM manager should be responsible for the state management of vTPMs (such as loading their secrets in the pTPM when the VM or container is starting) and even their migration. Because of this, the vTPM manager should have access to the pTPM interface to issue commands to it.

1.3. Paper structure

The rest of the paper is organised as follows. Section 2 introduces the concepts of TPM 2.0 architecture and features that are necessary for the understanding of our proposal An overview of the related research in the field of hardware-based trust applied to virtualised environments is presented in Section 3. In Section 4, we discuss the design of our TPM extension and the key features that enable the binding between a pTPM and one or more vTPM instances. We detail in depth the relationship between the physical and virtual platforms for key management and state protection in Section 5 and Section 6, respectively. Then, we present a low-level view of the extension commands and the envisioned functional mapping between a vTPM and a pTPM in Section 7, followed by the threat model of the target use case in Section 8. Then, we evaluate advantages and possible weaknesses in Section 9. Finally, we draw our vision of possible future works and our conclusions in Section 10 and Section 11 respectively.

2. Primer on TPM 2.0 capabilities

The TPM is a discrete cryptographic coprocessor that is already available in commodity machines and can serve as an enabling technology for several security workflows. Its specifications are developed by the TCG consortium, which comprises software companies, such as Microsoft that leverage the TPM to protect the BitLocker disk encryption key, platform manufacturers, that store boot measurements in the TPM for example, and several hardware vendors that are responsible for the actual implementation of the chip. Since its beginning, the TPM has included basic functions such as key generation, secure storage and reporting.

The first widely available TPM release was 1.1b, although the 1.2 version has been the most used for several years. To address the early privacy issues that arose from using a TPM, the 1.2 specification included a *privacy Certification Authority* (privacy CA) to assess that a key was generated by a TPM without disclosing the TPM instance it came from. Later, this will be enhanced with the development of new cryptographic protocols, such as Direct Anonymous Attestation (DAA) [3].

Platform configuration register (pcr). PCRs were defined in the TPM memory to store the integrity of boot time measurements (e.g. BIOS, bootloader). These registers could be securely reported by an identity key generated by the TPM so that an external entity could verify the integrity of the platform. The 1.2 release provided a standard interface to be implemented by all the vendors, which helped platform manufacturers to adopt the TPM device in their machines.

Non-volatile random-access memory (nvram). A TPM should include some non-volatile storage space that can store any type of data. A limited portion of the NVRAM is usually used to store the TPM's vendor certificate of the TPM endorsement key.

Cryptographic algorithms. Up to, and including, the 1.2 release, the TPM specification only supported the RSA, AES and SHA-1 cryptographic algorithms.

The latest TPM 2.0 specification significantly differs from the previous version, as TCG has redefined the internal structures, commands and algorithms (particularly cryptographic agility to allow for multiple ciphers). Nevertheless, the security capabilities of the device - described in depth as follows - have been kept intact and extended with new features.

2.1. TPM 2.0 key management

One tpm, three masters. The TPM 2.0 generates keys under different *key hierarchies*, namely the Endorsement, Platform, Storage (also known as Owner) and Null hierarchies [4]. Each of them is characterised by a *primary seed*, i.e. a large random number that is used by an internal *Key Derivation Function* (KDF) to create *primary keys*. Hierarchies differ by their controlling entity, authorisation mechanisms and when their seed is generated or reset.

More specifically, the Endorsement Primary Seed (EPS) and the Endorsement Key (EK) certificate are generated by the manufacturer of the TPM; the EPS can be replaced, by a random number generated by the TPM itself, after manufacturing but this invalidates the EK and its certificate, therefore the platform manufacturer usually disables this capability. The Platform Primary Seed (PPS) is under the control of the manufacturer of the computing platform (via the platform's firmware) that embeds the TPM, while the Storage Primary Seed (SPS) is controlled by the final owner of the platform, i.e. the end user. Because of its nature, the SPS (respectively the PPS) can be cleared and regenerated after platform manufacturing by its end owner (respectively the firmware); this subsequently invalidates all the keys created under the Storage hierarchy (respectively Platform hierarchy). Finally, the Null hierarchy is only used to generate ephemeral keys, hence its primary seed is not persistent. The hierarchies, except the Null hierarchy, can be independently disabled without affecting the use of the TPM by the other entities. In TPM 1.2, disabling the Storage hierarchy completely rendered unusable the device to all the TPM's users.

TPM 2.0 primary keys. Similarly to TPM 1.2, the latest specifications allow generating keys with different purposes, such as encipherment or digital signature. Encryption keys are the only allowed to be *parent keys* in a hierarchy, as they can encrypt – or *wrap* – child keys or objects: such key is named a storage parent. A primary storage parent is the root of a tree of storage parents and TPM objects, under the hierarchy whose seed has been used to create the primary storage parent. The primary keys differ from the ordinary keys as their *private* part – encrypted or not – never leaves the device. Ordinary keys can be used to encipher or sign external data and can be duplicated to different hierarchies (or even TPMs).

The most significant primary key, at least early in the life-cycle of a TPM, is the *Endorsement Key* (EK): its distinctive characteristic is that it is generated during the TPM's manufacturing and it is associated with an EK certificate signed by the TPM's vendor. The EK certificate is usually stored in the TPM's NVRAM for convenience. In TC, this is the way to establish initially the genuineness of a TPM. Due to privacy concerns, the EK for TPM 1.2 is a decryption-only key. Differently from the previous release, the TPM 2.0 allows the EK to be a signing key for TPMs of non-user devices (e.g. server, network router [5]). Although possible, it is not deployed and all TCG mechanisms use the decryption EK.

Another TPM 1.2 important key is the *Storage Root Key* (SRK): the SRK is the root of the only key, and object, hierarchy of a TPM 1.2. The SRK is replaced in TPM 2.0 by the primary storage parents that belong to any hierarchy and each hierarchy can possess multiple primary storage parents.

Given the limited amount of memory (volatile and non-volatile), a TPM must *load* keys and objects before using them. This is done by providing the public and private parts of an ordinary object to the TPM. By contrast, primary keys are deterministically generated inside the TPM, when needed, using the primary seed and a *template* [6] as arguments. This is a data structure that identifies the attributes of the object to be generated by the TPM. In fact, it must be provided at the creation

time of all types of objects; the particularity of primary objects in that they cannot be loaded, only re-created. The template includes the following information [7]:

- the object type, which may indicate a symmetric key, HMAC key, asymmetric key, or a data value;
- the name's algorithm, which identifies a hash algorithm used for computing the *Name* of the object (i.e. its unique identifier);
- the object attributes, a set of flags that determine the usage of the object and the rules to load it in the TPM;
- an authorisation policy to access the object;
- a unique value that may be used by the TPM at object creation process, as a user-provided argument.

In addition, the TPM allows customisation of the primary key by providing *sensitive* data. This also allows to set an authorisation value, i.e. a password, for the created object. The object attribute flags [7] can be either *set* or *clear* and form a bit-mask. They include:

- *fixedTPM*: if set, the object cannot be duplicated;
- *fixedParent*: if clear, the key itself can be duplicated. If set and *fixedTPM* is clear, the object could be migrated by virtue of one of its parents being duplicable.
- *restricted*: if set, the object (which is necessarily a key) can only perform cryptographic operations on structures of known format; for example, a *restricted signing* key cannot sign an arbitrary value, only data that has been hashed by the TPM itself.
- *sensitiveDataOrigin*: if set, the TPM will generate the sensitive data (e.g. the symmetric key). Otherwise, it is provided by the caller.
- *sign*: if set, the key can be used for signing; for a symmetric algorithm, this means performing an HMAC computation. For symmetric keys, this flag also controls the ability to encrypt.
- *decrypt*: if set, the key can be used to decrypt. Encryption is done by the public key for asymmetric algorithms, and by the same key for symmetric algorithms.

If the key creation succeeds, the TPM returns a 32 bit handle for the object that can be used for subsequent interactions (e.g. creation of a child key). A TPM handle is an identifier that the TPM uses internally to use the selected object. The TPM supports different handle types [7] that refer to specific object categories, the most relevant of which are:

- PCR registers;
- NVRAM indexes;
- permanent objects, such as hierarchies;
- transient objects, such as primary keys at creation;
- persistent objects, which have been loaded and made persistent in the internal memory of the device.

With respect to TPM 1.2, the latest specification offers a greater algorithm flexibility. The major differences are support for *Elliptic Curve Cryptography* (ECC) algorithms (e.g. ECC P256, ECC BN256), the SHA-2 family, and it enables regionalisation of the TPM, for example to use the SM9 Chinese cryptographic standards.

2.2. Platform attestation

Platform configuration registers. The TPM 2.0 includes one or more banks of PCRs (i.e. sets of internal registers) that are specific to a cryptographic hash function (e.g. SHA-1, SHA-256). The value of a PCR can be updated only via an *extend* operation. This operation concatenates the current value of the PCR with the new data and computes the hash over the result, before updating the PCR value with it. The new data is often a cryptographic hash, although the TPM specification does not enforce this format. Because of its nature, the value of a PCR depends on the order of the operation that extended value in the PCR.

All PCRs are initialised at platform boot, to all zeros or all ones. Commodity machines typically have 24 PCRs in each bank, although the exact number depends on the manufacturer. PCRs cannot store persistent data, and most of them cannot be reset without a reboot of the platform.[1] Because of their nature, PCRs are used to store measurements recorded during the life-cycle of a platform, so that they can be presented to an external entity as proof of the integrity of the system.

Measured boot. At platform startup, a *Core Root of Trust for Measurement* (CRTM) is responsible to calculate the digest of the next software being loaded in memory, and extend its value in a PCR. Then, the next software is executed and continues the measurement chain up to the *Operating System* (OS) kernel loading, storing the measurements in the available PCRs. This process, known as *measured boot*, allows to store measurements of the boot process (e.g. BIOS, boot-loader) in the PCRs so that they can be verified against known-good whitelists. The CRTM is the only software component of the architecture that must be trusted by default, as it cannot be measured or validated. Because of this, it is known as a *root of trust* and it should be shielded by the platform manufacturer against manipulations.

In addition to boot integrity measurements, the Linux kernel integrates the *Integrity Measurement Architecture* (IMA) [8] module to record software events occurring at runtime. These include binaries that are executed on top of the OS or files open for read, and they can be measured according to a policy that is configurable by the system administrator. All the digests measured by IMA are extended to the same PCR, so that the aggregated value of the measurements can be compared against a known-good value, which is typically computed by extending the white-listed digests of all measured software components with the same order.

Remote attestation (ra). TPM-based remote attestation leverages a specific operation, named *quote*. The TPM hashes a number of caller-selected PCRs that contain proof of the platform integrity, and signs the digest with an Attestation Key (AK), which is a restricted signing key. The signature, together with the PCR digest report and an anti-replay nonce – provided by the caller – represent the RA *Integrity Report* (IR) [9]. The IR is provided to a *verifier* by the TPM-equipped *attester* (the name of the platform being verified), so that the remote party can verify that it was not altered. Of course, this assumes that the AK came from an authentic TPM. To bridge that gap, different schemes have been defined in the literature to certify AKs. These include the aforementioned privacy CA, wherein a EK can be used to prove that an AK originated from a TPM without disclosing the TPM identity, and *Direct Anonymous Attestation* (DAA), an alternative scheme based on group signatures. TPM 2.0 also allows an AK to be generated under the Endorsement or Platform hierarchies and certified by the manufacturer. This approach is suitable for environments wherein privacy of the TPM identity is not a concern.

2.3. Secure storage

Sealed data object. Since its initial specification, the TPM device has provided means to securely store certain amounts of data. In TPM 2.0, a specific object type is defined for this purpose:

[1] To support a measured launch, some PCRs can be reset during the platform's runtime through special instructions.

the *Sealed Data Object* (SDO). This is achieved by a combination of *keyed hash* object type and the *sign* and *decrypt* flags both cleared, which means an object neither for signature nor for encipherment. Moreover, the *restricted* flag must be clear as well, as the SDO is generated from a data blob that is not provided by the TPM. In fact, the data blob is passed to the TPM as part of the *sensitive* area along with the template and the handle of the parent key.

The TPM 2.0 supports context management, which allows objects to be loaded into the device when needed. When a SDO is swapped out of context, it is stored encrypted with a symmetric key that can either be its parent key (if symmetric) or protected by the parent asymmetric key. The major drawback of storing sensitive information in a SDO is its size limitation. In fact, the specifications state that a *keyed hash* data type cannot exceed the limit of 128 B.

Nvram index. Given the strict limitations of SDOs, the TPM 2.0 specifications allow to store data in NVRAM indexes. An NVRAM index is an area of memory that can be used to store either TPM known data structures or unstructured data defined by the end user. The user must specify the size and attributes of the index before using it, and control access to the index via heterogeneous policies (e.g. shared secrets). Given the size of the NVRAM (in the order of KiB), this is a suggested approach to store large persistent data (such as digital certificates) in the TPM rather than SDOs. The NVRAM supports different types of indexes [4]:

- *ordinary*, which stores unstructured data blobs of arbitrary length;
- *counter*, which stores 64 bit long counters that can only incremented;
- *bit field*, which holds 64 bit that are clear when initialised and can only be set;
- *extend*, which is initialised with all zeros for a specific hash algorithm and can only be extended (as a PCR).

3. Related work

The original vTPM design was developed by Berger et al. [10], who focused on the 1.2 specification. The authors proposed that a vTPM manager, running in a privileged VM, had access to the underlying pTPM through the hypervisor. This managed the instantiation of vTPM instances and exposed them to other VMs through client drivers available in the virtual domains. The limitation of this approach is that the vTPM state and keys are fully managed in software in the privileged domain, hence they cannot provide the same assurance and tamper resistance than hardware devices. The authors also proposed an architecture for *deep attestation* where a vTPM identity credential, i.e. a vTPM EK certificate, was bound to that of a pTPM, i.e. an AK certificate. Compared to the original design by Berger et al. this proposal offers hardware-level protection of the private keys generated by the vTPM instance, so that they cannot be manipulated by an attacker that gains access to the hypervisor without proper authorisation.

The authors of this work developed an architecture to perform run-time integrity attestation of software run inside Docker [11] containers [12]. Although viable, this approach suffers from a performance overhead at the increase of virtual instances, as a single pTPM (based on version 1.2) is used to store measurements of both the host and all the containers running on top of it.

Shi et al. [13] defined a technique to protect vTPM secrets (in software) using a symmetric encryption key wrapped by a pTPM Storage hierarchy key. Their implementation uses QEMU/KVM as a means to expose the vTPM to the virtual environment. Compared to our approach, once the vTPM secrets are decrypted at VM instantiation, there is no interaction with the pTPM and all the vTPM objects reside in VM memory space, making them vulnerable to software attacks.

Hosseinzadeh et al. [14] propose two different architectures where one or more vTPM instances are exposed to containers and linked with a pTPM. The first relies on the vTPM instances to be integrated into the host kernel and to expose character devices that are attached to containers, thanks to the container runtime. Compared to [10], this design does not rely on the virtualisation engine (in the first case, the hypervisor) to manage all hardware accesses to the pTPM. A second proposal is similar to [10] in the sense that each vTPM is placed inside a separate container. Only this container has access to the hardware TPM and it exposes vTPM interfaces to the other containers through UNIX sockets or other communication channels. The second approach requires a daemon to manage vTPM instantiation, rather than the kernel as in the first scenario.

Wan et al. [15] discuss the limitations of traditional TPM design when applied to a virtualised environment. The TPM is originally designed to support a single host and cannot handle simultaneous access by multiple entities. The authors discuss alternative designs for vTPMs in literature, including the original proposal [10] and an alternative approach based on para-virtualisation of pTPM [16]. Compared to previous solutions, the latter tries to overcome the security issues of fully virtualised TPMs when exposed to VMs, although it does not overcome the limitations in the current TPM design.

The open-source community has recently proposed a software tool, named *Keylime* [17,18], that aims to bootstrap trust in a cloud environment. In this scenario, the vTPM is a software module that resides in the host machine and is exported to traditional virtual machines based on QEMU/KVM. The protection of the vTPM software is implemented via *Linux Security Modules* to enforce *Mandatory Access Control* (MAC) policies to prevent malicious users' activities.

Trusted Execution Technologies (TEEs) such as Intel *Software Guard Extensions* (SGX) [19], AMD *Secure Encrypted Virtualisation* (SEV) [20] and ARM *TrustZone* [21] have been proposed as a solution to secure software run in virtualised instances. Wang et al. [22] have proposed protection of a virtualised instance of TPM 2.0 (based on the open-source *libtpms* library) in a SGX *enclave*, a shielded memory area wherein only trusted code is allowed. These mechanisms are complementary to our proposed design, which reinforces them by securing secrets in a discrete component immune to attacks such as side-channel that are effective against TEEs.

More recently, the authors of [23] proposed an extension of the vTPM to support the SM cipher suites. The authors of [24] presented a Container Integrity Measurement (CIM) scheme which extends the chain of trust to bare-metal containers and virtual machine containers and fundamentally realises the integrity protection of containerised Virtual Network Functions (VNFs). This scheme heavily relies on the vTPM. Wang et al. [25] identified critical challenges for vTPM protection with SGX and proposed SvTPM, which is an SGX-based virtual trusted platform module, to provide vTPM run-time protection and strong isolation based on SGX. They also designed the solution to be robust to attacks such as NVRAM binding and vTPM snapshot rollback. A prototype is available and based on QEMU and KVM. Finally, the authors of [26] presented a solution to protect the integrity of VNFs using the vTPM.

Table 1 summarises the limitations of the works in the current literature and our advantages over them.

Table 1
Summary of the advantages of our solution against current literature.

Work(s)	Limitations	Our advantages
[10,14,24,26]	vTPM state and keys fully managed in software.	Hardware-level protection of the private keys generated by the vTPM.
[12]	Performance overhead as the virtual instances increase due to the single pTPM.	Hybrid software-hardware key management to avoid the pTPM as a bottleneck.
[13]	vTPM objects resider in VM memory spaces leaving room for software attacks.	Consistent interaction with the pTPM and TPM objects can be stored in hardware.
[15]	It does not overcome the security limitations of fully virtualised TPM in the current TPM design.	Our work is based on the latest TCG specifications.
[17,18]	The protection of the vTPM software is done using MAC.	Hardware-level protection.
[19–22,25]	Orthogonal works and technologies that can extend and reinforce our approach.	–

4. vTPM-to-pTPM cryptographic binding

The primary goal of our work is to bind a vTPM to a pTPM, ensuring that cryptographic keys generated in each hierarchy are protected by a hardware platform, rather than software. Most of the TPM objects are ordinary objects, which are protected by a Storage Parent Key hierarchy that is ultimately anchored in a primary key. The primary keys represent the roots of each hierarchy, and they are generated from primary seeds as specified in Section 2. Hence, our proposal aims to protect the generation of vTPMs seeds with the underlying pTPM, and then extends the protection to primary keys and ordinary objects. It is to be noted that seeds should be protected against vTPM failures, as they are required to generate primary keys under each vTPM hierarchy (and hence, to load keys and unseal secrets that were stored by the vTPM before failure). Because of this, a management entity, the vTPM manager, should run at the host level to keep records about vTPM instances and their associated data (such as seeds).

4.1. Limitations of conservative approaches

Several approaches for the pTPM-to-vTPM binding have been considered in our research. They aim at adapting the existing specifications to the virtualised environment and are described thereafter.

4.1.1. vTPM seed generation through pTPM at manufacturing time

The most conservative approach would require the pTPM to generate primary seeds at vTPM instantiation time, using the device hardware *Random Number Generator* (RNG).

This proposal would have the benefit to leverage a hardware RNG, that is more secure than a software RNG used by a standard vTPM implementation. Once the seeds are generated, they should be stored by the vTPM software in the proper data structures for their run-time. In addition, these seeds could be sealed by the pTPM as SDOs, using an encryption key under a persistent hierarchy (e.g. the Storage hierarchy) so that they can be recovered in case of vTPM failure. Moreover, they could be made persistent by storing them in the NVRAM. In this approach, a vTPM manager should keep information about the SDOs handles and the related vTPM instances, so that the seeds can be re-injected in them after a restart. An advantage of this approach is that the pTPM and the vTPMs are independent from each other (other than for the SDO protection), making the vTPMs more scalable in large virtualised infrastructures. Moreover, the vTPMs are easily migratable to other instances as their seeds (the roots of the hierarchies) are managed in software.

The major drawback of this approach is that the vTPM loads the primary seeds unencrypted in its memory so that it can manage the key hierarchies. Even if the seeds were encrypted on disk, an in-memory attack could still get access to clear-texts, compromising the vTPM instance.

4.1.2. Multiplexing of pTPM hierarchy root keys to vTPM instances

An alternative approach would be to leverage the customisation options at object creation given by the TPM 2.0 specifications. More specifically, a vTPM manager could leverage the *unique* value in the public template of the primary keys to generate unique root keys in each hierarchy of the pTPM. These unique values could be generated through the pTPM RNG and then associated to each vTPM instance by the manager.

This approach would improve the previous proposal, as the vTPM key hierarchies are managed exclusively by the pTPM. Both the seeds and the primary keys are never exposed in software, achieving hardware-level security. pTPM context management would allow to manage the increasing number of vTPM keys, although this design requires careful management of the unique values so that key hierarchies are separate.

This approach suffers from the limitation of generating both pTPM and vTPMs primary keys from the same hardware-based primary seeds. An undesirable side effect is that the life-cycles of the pTPM hierarchies are now tied to all the vTPMs using the platform.

4.2. Our proposal: pTPM 2.0 virtualisation extension

We address the limitations of the existing approaches and aim to overcome them by proposing a *virtualisation extension* to the TPM 2.0 specifications. Our goal is to extend the TPM 2.0 design as effectively as possible to enable management of vTPM identities and their life-cycles in a pTPM-equipped host, including migration of the virtual instances. Moreover, we aim to retain the hardware-level security of a pTPM even in the virtualised environment, at least for the core identity elements (such as seeds and primary keys) that should never leave the device unprotected. One should note that we can use the pTPM duplication mechanism to assist the migration of the vTPM.

In our approach, the pTPM interface is extended to enable hardware protection of vTPM primary seeds, wrapping of vTPM primary keys and creation of child objects, as described in the following Section 5. Moreover, the pTPM is used to wrap cryptographic keys utilised for encryption and decryption of the persistent state of vTPM instances during their life-cycle, as depicted in Section 6. The comprehensive list of added commands to the pTPM interface and the required mapping between the vTPM and pTPM interfaces are presented in Section 7.

While this proposal requires a new semantic to be shared and understood by both the pTPM and vTPM, one of the ancillary goals is to minimise the impact on the vTPM's user or a *Trusted Software Stack*. This is achieved by designing the new semantic required by this proposal to use existing but unused attribute bits.

Table 2
vTPM primary seed object attributes.

Sign	Decrypt	Restricted	Object use
0	0	0	Sealed Data Object
0	**0**	**1**	**Virtual Primary Seed**
0	1	0	Decrypt only, not parent key
0	1	1	Parent key
1	0	0	Quote, sign, certify, HMAC
1	0	1	Quote, sign, certify (TPM structures)
1	1	0	Sign, decrypt, not parent key
1	1	1	Not supported

5. vTPM key management

This section describes the design changes that are envisioned on the pTPM 2.0 interface to support hardware-bound key generation and usage. These are to be implemented by the vTPM software so that calls to object management commands (e.g. creation of a virtual primary key) are mapped to new commands exposed by the pTPM interface.

5.1. vTPM primary seeds

To achieve our goal, we first propose the inclusion of a new TPM object: the *Virtual Primary Seed* (VPS). This is constructed as a variation of a standard SDO, in the sense that it leverages the existing *keyed hash* object type with a particular combination of the object attributes bit-mask. The adoption of an existing object type has a smaller impact on specifications, rather than defining a completely new type (which would affect both the internal structures of the TPM and its commands). Each VPS is wrapped by a parent pTPM key, hereby named *Seed Wrapping Key* (SWK), whose hierarchy is used to identify the VPS hierarchy. Concerning its object attributes, there are two possible approaches that derive from SDO attribute selection, which requires all *sign*, *decrypt*, and *restricted* to be clear:

- set the *restricted* bit that is forbidden in the current specifications in case of a SDO, as shown in Table 2;
- set the *sensitiveDataOrigin* flag that is clear in the standard SDO (as the data are generated outside of the TPM).

Although both approaches are viable, we choose the first option as it semantically forbids the pTPM to use the VPS for operations on unknown data structures or external data blobs. Also, as explained in the next paragraph, the VPS value may be generated by the pTPM itself; *sensitiveDataOrigin* does not make sense in that case.

The seed generation can leverage the internal pTPM RNG, as in previously proposed approaches. An inherent limitation of our proposal is that the seed must fit the size limitation of a standard SDO, to avoid breaking the existing constraints defined in specifications [7]. This constraint is currently of 128 B, as previously mentioned, which is superior to the actual size of primary seeds used by current TPM 2.0 simulators, i.e. 64 B.

The SWK is used to wrap the primary seed so that its context can be stored in the vTPM without disclosing the seed in clear text. At vTPM instantiation, the *virtual Endorsement Primary Seed* (vEPS), *virtual Platform Primary Seed* (vPPS) and *virtual Storage Primary Seed* (vSPS) must be generated via the underlying pTPM. The NULL hierarchy seed can be either generated via the pTPM RNG or fully implemented in software. Fig. 2 shows the relation between the pTPM per-hierarchy SWKs and multiple VPS objects associated to different vTPMs.

After generation, the vTPM software is responsible for loading the VPS in the underlying pTPM, when required for primary key generation, so that the seed value is decrypted in the pTPM protected memory and a transient handle is returned to the vTPM. If needed, the seed object can even be made persistent to the pTPM NVRAM by using standard commands. Moreover, the vTPM primary seed cannot be unsealed as standard SDOs, as the unseal operation is designed to fail if either *sign*, *decrypt* or *restricted* flags are set [6]. At vTPM runtime, the instance only knows the handle of each VPS, while the pTPM is responsible for protecting and accessing the seed values.

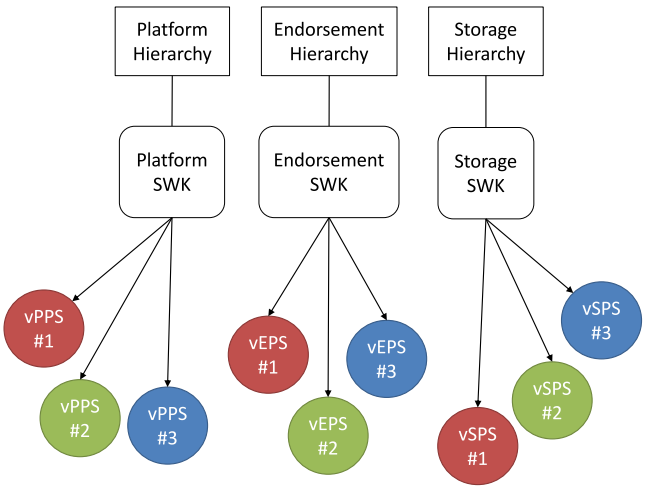

Fig. 2. vTPM primary seed object definition within the pTPM hierarchies.

The VPS can be either made fixed to the pTPM or duplicable by properly setting the *fixedTPM* and *fixedParent* object attributes. This flexibility enables a vTPM owner to decide whether certain vTPM-equipped VMs or containers should be pinned to a specific host or be duplicatable.

5.2. vTPM primary keys

In our design, virtual primary keys share similar security properties with their physical counterparts. In fact, the key can be configured such that the private area never leaves the pTPM, as the virtual primary key is automatically loaded in the pTPM context when generated. A handle of the primary key is returned by the pTPM as it is managed by the physical device and is stored in the vTPM context to be used afterwards. Similarly to the pTPM primary keys that use a hierarchy handle, the vTPM key generation function receives a vTPM hierarchy handle, in addition to the public template, but the vTPM needs to translate this into using the VPS and the pTPM. Because of this, the vTPM seed must be previously generated at vTPM instantiation, stored by the software instance or loaded in the pTPM. Depending on the VPS origin, the hierarchy of the primary key is derived. The high-level sequence of steps occurring in the pTPM to generate a vTPM primary key is defined as follows:

1. verify that object attributes in the input public area are valid (e.g. the *restricted* bit is set);
2. verify that the input handle belongs to a loaded vTPM seed object, and get its sensitive area (i.e. the random seed);
3. validate the sensitive area values;
4. create the primary object by running the pTPM KDF with the vTPM seed as an input parameter;
5. return the transient handle of the primary object.

Although sensitive areas of primary keys never leave the device, they can be still migrated in case a vTPM is moved to another

pTPM-equipped instance as they can be deterministically regenerated by a KDF if the same public key template and sensitive area are provided, along with the VPS. Because of this, a vTPM primary key is duplicable if the *fixedTPM* and *fixedParent* object attributes of the corresponding primary seed are clear.

5.3. vTPM hardware-bound and software child objects

When designing a vTPM architecture for a cloud environment, flexibility should be considered in order to offer different trade-offs between performance and security depending on the level of assurance required by a certain workload. Because of this, we envision that the vTPM should support the creation of either pTPM-protected child objects (whose sensitive area does not leave the pTPM unencrypted) and fully virtual objects and enable a vTPM user to attest to the object protection.

The first option does not require any changes to the pTPM interface, as it is invoked for the generation of a child object starting from a pTPM object handle. The vTPM interface internally invokes the pTPM object creation command and retrieves the object public and private areas, the latter being encrypted with the pTPM protected parent key. Then, when the object is loaded by the vTPM user, the request is forwarded to the pTPM and the vTPM returns the object handle to the user.

A vTPM *software object* is an object whose sensitive area can be accessed by the vTPM instance. No or minimal interaction with the pTPM should be in place to achieve this purpose, which makes our architecture suitable for applications that would leverage the vTPM for its cryptographic operations rather than hardware security. In general, software key management is expected to achieve higher performance than pTPM-bound operations, which are limited by the TPM access broker, resource manager, and inherently limited computing resources. However, fully virtualised keys are less protected than hardware-bound objects, hence additional security mechanisms that protect applications' memory should be in place (e.g. SGX enclaves to secure the vTPM memory, as developed in [22]).

The pTPM does not need to comprehend the semantics of vTPM software objects, as they do not require interaction with the physical platform. In turn, the vTPM interface should be aware of the generation of either hardware-bound or software objects. This awareness can be obtained by introducing a new object attribute flag, hereby named *pTPMCreated*, to be leveraged by vTPM object creation functions to discriminate whether the pTPM should be invoked or not.

The TPM 2.0 specification [7] notes that several bits of the object attribute bit-mask are reserved for future additions, hence any of them is suitable for our purpose. The *pTPMCreated* object attribute flag has the following meaning:

- **SET (1):** the object sensitive area is protected by the underlying pTPM only;
- **CLEAR (0):** the object sensitive area (i.e. private part) may exist in plaintext in the vTPM.

This attribute would also apply to the VPS object, allowing the vTPM to be instantiated fully in software. Although this approach lowers the security level of the instance, it enables a more flexible deployment of our vTPM architecture in a CSP infrastructure. In case of hardware-bound VPS objects, they can be certified directly from the pTPM. One should also note that the *pTPMCreated* object attribute is only meaningful for a vTPM; pTPMs can either ignore the *pTPMCreated* object attribute, or mandate that it is set when they support the proposed vTPM extension.

We assume that all the children of a vTPM software object may not be bound to a pTPM, hence some may have *pTPMCreated* clear. This constraint is made necessary to simplify the process required

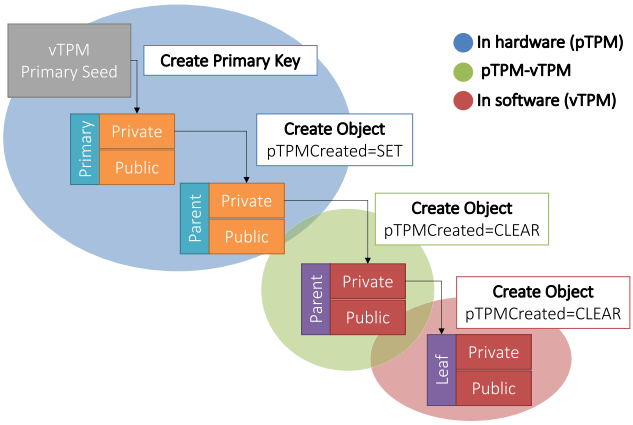

Fig. 3. vTPM software object hierarchy.

to generate a vTPM software object. To create any object, the vTPM requires the parent key to be loaded beforehand. This is not trivial in case a vTPM object – with *pTPMCreated* set – is wrapped by a parent key that was not generated by the pTPM. Because of this, a pTPM would only accept the parent key *pTPMCreated* bit to be set whenever it is invoked by the vTPM for object or key generation.

The generation of a software object from a hardware-bound parent key, as depicted in Fig. 3, is challenging from the research standpoint. In fact, the first software object in each hierarchy has a parent with *pTPMCreated* set, hence its parent must be loaded in the pTPM first. Because of this, the intermediary stage between hardware-protected keys and fully virtualised objects still requires an interaction with the pTPM by design. In this regard, we envision two possible approaches to generate the first software object in the vTPM hierarchy, described as follows.

5.3.1. Software object as unsealed pTPM SDO

The TPM 2.0 command specification [6] requires the SDO to be generated under a parent key so that its value – encrypted with a symmetric – can be exported unencrypted. So, a viable approach to software objects with pTPM protected parent would be to store their sensitive area as pTPM SDO payloads, regardless of the object type requested on the vTPM interface. This would require the vTPM software to convert the object creation to a *keyed hash* object type and to generate the object sensitive area in software so that it can be presented to the pTPM as standard SDO. The resulting private and public areas would be returned by the vTPM by presenting the original object template rather than the pTPM-generated SDO. At loading time, the vTPM would execute the following steps:

1. convert the object public area template to a SDO;
2. load the public and private areas in the pTPM, retrieving the SDO handle;
3. unseal the sensitive area of the object by its handle in the pTPM;
4. store the sensitive area of the object along with the original object template in the vTPM memory.

The major drawbacks of this approach are the extensive amounts of mapping between actual object type and SDO both at creation and loading time and the size limitation of the SDO sensitive area (fixed to 128 B in the TPM 2.0 specification). Because of this, the vTPM may need to store the elements required to re-generate a key (e.g. RSA prime factors) rather than the TPM private object structure, whose size may exceed the SDO size limitation (e.g. the RSA-2048 private object size on disk is 224 B).

5.3.2. Software object as pTPM duplicable object

An alternative approach is to leverage the TPM 2.0 key *duplication* capabilities [4]. In fact, the TPM allows to move an object to another parent if its attributes do not prohibit duplication and its authorisation policy explicitly allows it. Creation of a duplicate uses two encryption phases. The first is used to apply an *inner wrapping* starting from a symmetric key shared between the original and the duplication platforms. The second, named *outer wrapping*, is used to encrypt the object using the algorithms of the new parent key. The result of the duplication is the private area of the object to be loaded by the new parent [7].

However, this process would require to generate a vTPM software parent key alongside the pTPM-protected hierarchies so that the software object can be moved to it. Moreover, if a new parent key was provided, the software object could not be loaded again under the former parent key. Because of this, this approach could leverage a particular form of key duplication that is made possible by the TPM. This requires the following properties to be met when the duplication command is issued:

- the object attributes *fixedParent* and *encryptedDuplication* are 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 to duplicate the object in clear-text by bypassing both outer and inner wrappings, achieving the same result as the unseal operation. At loading time, the vTPM would execute the following steps:

1. load the public and private areas in the pTPM, retrieving 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 memory.

The advantage of this approach is to leverage standard TPM structures and commands for object creation, without the need of mapping the vTPM requested object to a pTPM different type. A possible disadvantage is the management of the object authorisation policy for duplication, as this command requires the vTPM to enable a policy session. Moreover, the vTPM should present a different authorisation policy than the pTPM object, which creates a policy mapping task in the vTPM. Although both approaches are viable, the solution based on object duplication does not suffer from object size limitations as the SDO-based alternative. Moreover, it does not require multiple mappings between vTPM and pTPM object types hence it may perform better in a real world scenario.

6. vTPM state protection

This section describes the vTPM state composition and the protection mechanisms that can be enabled via the underlying pTPM thanks to the vTPM manager. These target the protection of the virtual instance state against manipulations when it is stopped or restored during its life-cycle.

6.1. vTPM persistent state

The vTPM instance is composed of a set of handles referencing the vTPM's objects. These have different natures depending on their relationship with their underlying pTPM:

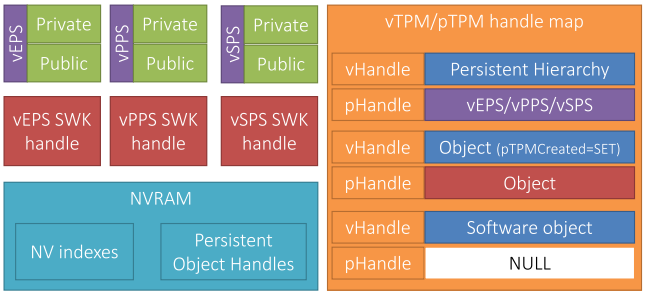

Fig. 4. vTPM persistent state structure.

- hardware-bound reserved handles, such as persistent hierarchy structures, which specify the VPS objects (with their authorisation and policies);
- pTPM-created object handles, such as primary keys and hardware-bound child objects;
- vTPM software object handles:
- other internal vTPM handles required for its functions.

In our design, the vTPM state includes a handle map that maps each entry, i.e. a 32 bit number, to a certain data structure (specified as a union, so that the value can be of different types). Each entry refers to a vTPM virtual handle, called a vHandle, while the value is mapped to a pTPM object (through its pTPM handle, called a pHandle). More specifically, persistent hierarchy handles should be mapped to VPS handles in the pTPM; objects generated by the pTPM should be mapped to the actual hardware handles so that they can be referenced when interacting with the physical device; vTPM software objects should be mapped to a *null* value to specify that they are persisted in vTPM state only.

The vTPM permanent state consists of the following data, as depicted in Fig. 4:

- public and private areas of the VPS objects, as returned by the pTPM (for a primary object, there is no private area);
- the handles of the SWK keys needed to load VPS objects;
- the NVRAM content, which includes user-defined indexes and objects persisted in the vTPM;
- the aforementioned handle map, including the handles of vTPM primary seeds, as loaded in the pTPM, along with other pTPM-protected objects and software objects.

Given its size, this amount of data cannot be protected by the pTPM as a SDO, hence the vTPM creates a primary encryption key under its Platform hierarchy for state encryption when the instance is stopped. Given its template (which should be defined at instantiation time), the key can be re-generated at each restore of the vTPM instance, so it does not have to be stored permanently by the pTPM. In this regard, the template should request object authentication and authorisation policies so that entities other than the vTPM instance (or a vTPM manager) cannot load the key.

6.2. vTPM initialisation

The vTPM manager is the component that starts vTPM instantiation in the host platform, before presenting it to any virtualised instance (e.g. by exposing a character device interface to a virtual machine). Before any instantiation, the manager should ensure that three parent keys are loaded in the pTPM, one for each permanent hierarchy (i.e. Platform, Endorsement and Storage). These must be defined as storage keys, so they can only be used to wrap the VPS objects (whose type is known to the pTPM). These keys can either be primary or child objects in the pTPM:

- if primary key, it is not necessary to store them at the physical platform shutdown as they can be re-generated each time by providing the same template;
- if child key, the vTPM manager should store their public and private parts, along with the references to their parents, so that they can be reloaded.

From a performance perspective, loading an existing child key is faster than re-generating a key (e.g. in the case of RSA, the operation lasts in the order of tens of seconds).

The following operations should be performed by the vTPM manager when a new vTPM is instantiated:

1. the vTPM manager initiates vTPM instance by loading the software module and configuring the vTPM's handles of the vEPS, vPPS, and vSPS SWK keys;
2. later, the vTPM instance receives the *init* command from the entity using the vTPM, as defined in the specification [6];
3. the vTPM instance requests, to the pTPM, creation of the vEPS, vPPS, and vSPS objects by passing the handles of the corresponding SWK keys, and stores their public and private parts in its state;
4. the vTPM software loads the vEPS, vPPS, and vSPS objects in the pTPM and stores their handles in its state;
5. if the vTPM initialisation succeeds, the vTPM generates a primary symmetric encryption key under its Platform hierarchy by invoking the pTPM interface with the vPPS handle, and stores the resulting handle internally. This key is called the *vTPM State Protection Key*(VSPK).

The VSPK generated in the last step of the vTPM initialisation is used for the storage and restoration operations of the vTPM, described as follows. The public template of the VSPK should be known to the vTPM instance so that it can create the key at first instantiation and even re-generate it at each restore operation. Because of this, the vTPM manager must define the template and pass it to the vTPM instance. Moreover, the template should enable the vTPM instance to only decipher its own state when restored. To do so, the unique field of the public template can be used to generate a separate VSPK for each vTPM instance under the same pTPM Platform hierarchy. In order to protect the state against attackers, an authorisation policy can be specified so that the use of the VSPK is gated by a policy on the platform state (e.g. the values of certain PCRs).

6.3. Storage and restoration of vTPM instance

The core logic to store the vTPM instance is included in the *shutdown* command, as defined in the specification [6], whereas the state is restored as part of the *init* command whenever an encrypted state blob has been provided by the vTPM manager during instantiation.

At vTPM shutdown, the state data to be encrypted does not include the primary seed public and private areas, as they are already wrapped by the pTPM parent keys. The other state elements, i.e. the NVRAM content and the handle map, are encrypted with a symmetric algorithm by the pTPM (e.g. AES 128) and the resulting data blob is presented to the vTPM manager. In fact, the pTPM can be utilised to encipher arbitrary data via symmetric encryption when the instance is stopped and to decipher it when restored.

At vTPM initialisation, 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. This replaces step (3) detailed in 6.2: the VSPK is re-generated by the vTPM instance, under the vTPM Platform hierarchy, using the known template; the key is then used by the pTPM to decipher the encrypted state blob. Since the VSPK is a vTPM primary object, it is recreated from the vPPS and cannot be stored anywhere (the pTPM never returns the private parts). Once the state blob is decrypted, the vTPM restores the persisted objects. Moreover, for each of the hardware-protected vTPM objects, the corresponding NVRAM Persistent Object index must be updated so that it reflects the index in the pTPM NVRAM. This update is needed as vTPM persistent objects are not actually persisted in the pTPM once the virtual instance is stopped, hence they have to be re-loaded at the vTPM state restore.

Considering the proposed design for vTPM keys and state management, new TPM commands, for the pTPM, need to be introduced to enable that architecture and are described in Section 7.

7. pTPM virtualisation commands and mapping to vTPM

This section is divided in two part. Firstly, the new commands that a pTPM needs to support are presented. Secondly, the mapping between the vTPM commands to the underlying pTPM commands are detailed. Finally, the impact of the new commands to the Trusted Software Stack (TSS) is analysed.

7.1. pTPM virtualisation commands

Table 3 describes the virtualisation commands to be supported by the pTPM 2.0 specifications. These are internally leveraged by the vTPM software whenever particular functions are executed on the vTPM interface.

In particular, the `TPM2_VIRT_CreateSeed` and `TPM2_VIRT_LoadSeed` are designed as specialised versions of standard `TPM2_Create` and `TPM2_Load` commands, that are used respectively to generate an object (not a primary key) and to load it in the pTPM. Differently from the standard `TPM2_Create`, `TPM2_VIRT_CreateSeed` internally checks that the object attributes in the input public template are valid. More specifically, `sign`, `decrypt` should be `CLEAR`, and `restricted` should be `SET` as presented in 5.1. Then, the RNG function would be run to generate a random number (whose number of Bytes is provided as a parameter by the vTPM instance). The `TPM2_VIRT_LoadSeed` performs the same internal operations as `TPM2_Load` so that the transient object handle is returned after its private and public part have been loaded in the pTPM, and additionally checks that the provided object has the proper attributes of a vTPM seed.

The `TPM2_VIRT_CreatePrimary` command shares similarities with the `TPM2_CreatePrimary` command, which returns a primary key handle starting from an hierarchy permanent handle (which in turn defines the hierarchy of the key itself) and the public template. Differently from the standard command, `TPM2_VIRT_CreatePrimary` checks that the input handle belongs to a loaded vTPM seed – and not a persistent hierarchy internal handle – by inspecting its object attribute. Moreover, it uses the sensitive area of the vTPM seed as input of the internal `DRBG_InstantiateSeeded` KDF so that the primary object is generated by the pTPM hardware.

To protect the vTPM state when the vTPM is stopped, `TPM2_VIRT_StoreState` and `TPM2_VIRT_RestoreState` internally perform the symmetric encryption and decryption operations of the standard `TPM2_EncryptDecrypt2` function, the

Table 3
pTPM 2.0 virtualisation commands.

Usage	Command	Input	Output
Primary seed creation	TPM2_VIRT_CreateSeed	SWK handle, VPS template, VPS size (B)	VPS private/public areas
Primary seed loading	TPM2_VIRT_LoadSeed	VPS private/public areas	VPS handle
Primary key creation	TPM2_VIRT_CreatePrimary	VPS handle, public key template	Primary key handle
State encryption	TPM2_VIRT_StoreState	Clear-text state blob, VSPK handle	Encrypted state blob
State decryption	TPM2_VIRT_RestoreState	Encrypted state blob, VSPK handle	Clear-text state blob

Table 4
vTPM command mapping to pTPM extension.

vTPM function	vTPM command(s)	pTPM command(s)
Initialisation	_TPM_Init	[TPM2_VIRT_CreateSeed, TPM2_VIRT_LoadSeed, TPM2_VIRT_CreatePrimary]
Shutdown	TPM2_Shutdown	TPM2_VIRT_StoreState
Restart	_TPM_Init + Encrypted state blob	[TPM2_VIRT_CreateSeed, TPM2_VIRT_LoadSeed, TPM2_VIRT_CreatePrimary, TPM2_VIRT_RestoreState]
Primary key creation	TPM2_CreatePrimary	TPM2_VIRT_CreatePrimary
pTPM-protected object creation	TPM2_Create (pTPMCreated=SET)	TPM2_Create
pTPM-protected object loading	TPM2_Load	TPM2_Load
Software object creation (with pTPM-protected parent)	TPM2_Create (pTPMCreated=CLEAR)	TPM2_Create
Software object loading (with pTPM-protected parent)	TPM2_Load	[TPM2_Load, {TPM2_Duplicate, TPM2_Unseal}]

preferred command in TPM 2.0 specifications to perform symmetric encipherment or decipherment. They require that the key provided as input is of type TPM_ALG_SYMCIPHER. As mentioned previously, the state protection key is generated from the pTPM (using the TPM2_VIRT_CreatePrimary function) as part of the vTPM instantiation.

7.2. vTPM to pTPM virtualisation commands binding

The new pTPM command set is leveraged by the vTPM software whenever particular functions are executed on the vTPM interface. The mapping between vTPM commands and the underlying physical platform is reported in Table 4, which includes functions that require an interaction with the pTPM. All other vTPM functions are not expected to differ from the existing specified commands [6]. In particular, vTPM initialisation is mapped to both TPM2_VIRT_CreateSeed and TPM2_VIRT_LoadSeed functions, and creation of the state encryption primary key via TPM2_VIRT_CreatePrimary. During vTPM shutdown, the TPM2_VIRT_StoreState command is issued to encrypt the vTPM state and present it to the vTPM manager. At vTPM restart, the _TPM_Init command internally restores the state via TPM2_VIRT_RestoreState after having re-generated the state encryption key.

The creation and loading of pTPM-protected objects is performed by forwarding the standard TPM2_Create and TPM2_Load functions to the underlying pTPM. The creation and loading of software objects, in case of a pTPM-protected parent, can follow either the TPM2_Unseal or TPM2_Duplicate strategies, as discussed before. Table 4 lacks the description of functions to manage objects with software parents (i.e. parent keys with pTPMCreated clear), as they are not mapped to the pTPM. At this point of the hierarchy, the vTPM objects only reside in the software memory and the vTPM must behave accordingly with the TPM specification.

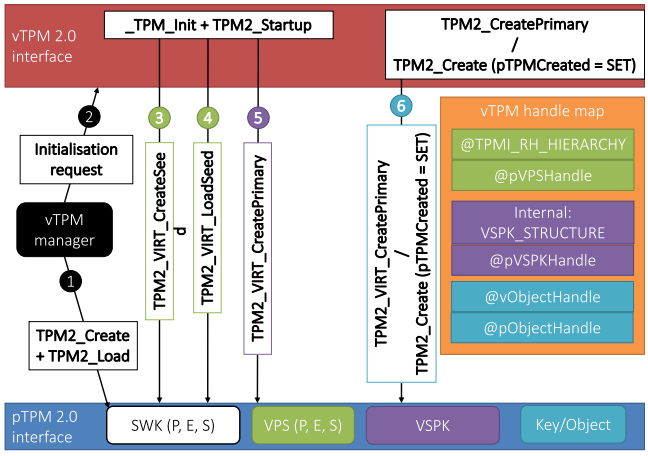

Fig. 5. vTPM-to-pTPM command mappings for initialisation and key generation.

7.2.1. vTPM initialisation with hardware binding

Fig. 5 describes the steps required at each vTPM instantiation whenever an hardware binding is required, and no saved state exists for the vTPM. In summary, the pTPM needs to create the SWK before creating the vTPM seeds. Once the vTPM seeds are loaded in the pTPM, hardware-bound vTPM primary objects can be created. The actors involved in this interaction are the vTPM manager, the pTPM 2.0 and the vTPM, performing the following operations:

1. the vTPM manager queries the pTPM interface to create and load the SWK keys, one for each persistent hierarchy, by issuing TPM2_Create and TPM2_Load commands;
2. the vTPM manager issues an initialisation request, which instantiates a vTPM instance and passes the SWK handles

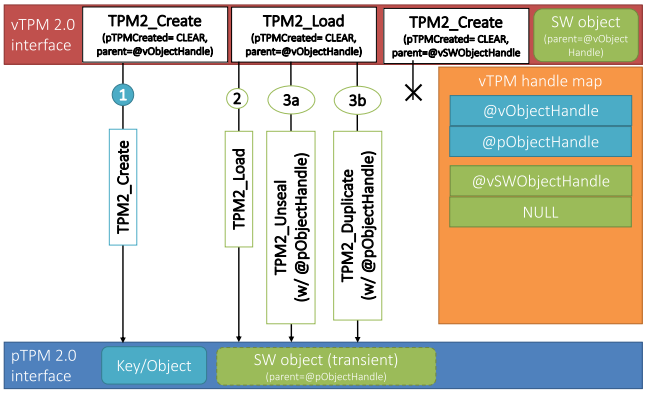

Fig. 6. vTPM-to-pTPM command mappings for software objects.

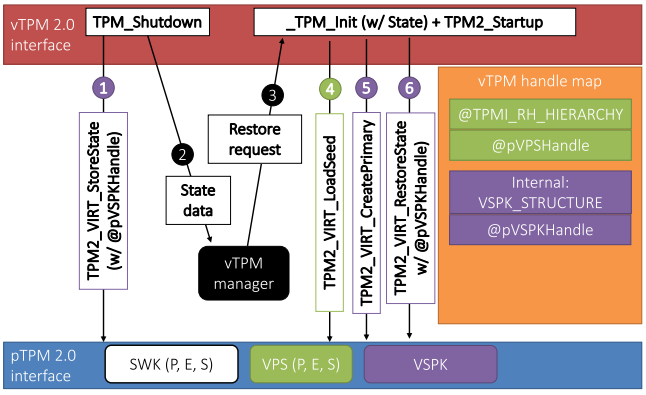

Fig. 7. vTPM-to-pTPM command mappings for state store and restore.

to it along with all the configuration required for its startup (e.g. the VSPK public template);
3. the vTPM instance receives the `_TPM_Init` signal from the platform, which in turn triggers the `TPM2_VIRT_CreateSeed` command for each persistent hierarchy so that the VPS objects are created;
4. the vTPM instance queries the pTPM `TPM2_VIRT_LoadSeedcommand` for each VPS object, storing its `pVPSHandle` handle in its handle map (corresponding to a `TPMI_RH_HIERARCHY` hierarchy reserved handle);
5. the vTPM instance queries the pTPM interface to generate the VSPK via the `TPM2_VIRT_CreatePrimary` command and stores the `pVSPKHandle` handle in the handle map, corresponding to an *internal* `VSPK_STRUCTURE` object (that is not part of the standard TPM 2.0 handles); at the end of this step, the vTPM `TPM2_Startup` command returns with success so that the vTPM user can finally use the device;
6. the vTPM user can generate its own primary and child objects by issuing either `TPM2_CreatePrimary` or `TPM2_Create` functions; in case an hardware binding is required for the object, the corresponding function is issued to the pTPM interface (either `TPM2_VIRT_CreatePrimary` for primary objects or `TPM2_Create` for hardware-bound child objects); each hardware-bound object introduces two different handles in the handle map, i.e. the vTPM `vObjectHandle` and pTPM `pObjectHandle` specific handles.

The initial step in the workflow may be performed by the vTPM manager at startup or first vTPM provisioning, so that all the vTPM instances are protected by the same SWK keys.

7.2.2. Creating a software object with a hardware-bound parent in a vTPM

Fig. 6 presents the interaction between the vTPM and the underlying pTPM whenever a software object is created starting from a pTPM-protected parent. The new vTPM software object is created, whose sensitive part (e.g., the plaintext asymmetric private key) is available directly in the vTPM, with the following commands:

1. the vTPM user issues a `TPM2_Create` command by specifying the *pTPMCreated* bit clear and passing the `vObjectHandle` parent handle, that in turn is mapped to a `TPM2_Create` command to the pTPM that leverages the object handle in the pTPM, i.e. `pObjectHandle`; the object is created by the pTPM, which can successfully load the private part of its parent object and returns both the private and public areas of the newborn; this may either be a SDO from the pTPM perspective, regardless of its actual type, or a duplicable object that retains its original type depending on the strategy selected from Section 5.3;
2. the vTPM user issues a `TPM2_Load` command on the vTPM interface, which is forwarded to the pTPM so that the newborn software object is loaded in the pTPM interface;
3. the vTPM interface issues one of the following commands to the pTPM interface, as discussed in Section 5.3:
   (a) `TPM2_Unseal` in case the software object is mapped to a standard SDO by the pTPM interface;
   (b) `TPM2_Duplicate` in case the software object is duplicable without encryption and a *null* parent.

   In both approaches, the sensitive area of the object is returned to the vTPM interface so that it can be stored in its memory.

Once the software object has been successfully loaded, it can be evicted from the pTPM memory. The subsequent vTPM commands involving this object do not leverage the pTPM commands, as the object `vSWObjectHandle` handle is mapped to a `NULL` value in the handle map. This means that the vTPM itself holds the sensitive area of the software object.

7.2.3. Hardware-bound vTPM state management

The interaction between the vTPM manager, the vTPM instance and the pTPM for storing and restoring the virtual instance is depicted in Fig. 7. During the vTPM shutdown, its state is encrypted by its VSPK, and stored by the vTPM manager. When the vTPM is started again, it creates its VSPK in order to decrypt and restore its state as follows:

1. the vTPM receives a `TPM_Shutdown` command (either by the virtualised environment wherein it is running or an hypervisor-level entity) and issues a `TPM2_VIRT_StoreStatecommand` towards the pTPM; this encrypts the vTPM state binary representation with the VSPK;
2. the vTPM presents the encrypted state to the vTPM manager for storage along with information for its subsequent restore (e.g. the VPS private and public parts);
3. at some point, the vTPM manager restores the vTPM instance by instantiating a new instance and passing to it the encrypted state blob along with its contextual information, such as the VSPK public template;
4. the vTPM instance receives a `_TPM_Init` command, with a saved vTPM stat, that internally triggers a `TPM2_VIRT_LoadSeedcommand` for each VPS object (so that primary seeds are not re-created from scratch);

5. the vTPM instance issues a `TPM2_VIRT_CreatePrimary` command to the pTPM by passing the VSPK template so that the exact same key is re-generated and its `pVSPKHandle` is restored in the handle map;
6. the vTPM instance issues a `TPM2_VIRT_RestoreState` command to the pTPM by passing the encrypted state and the VSPK handle, so that the cipher-text is decrypted.

Finally, the vTPM instance returns from the `TPM2_Startup` command, so that it can receive commands on its client interface.

7.3. Exposing the virtualisation extension in the trusted software stack

From a regular vTPM or pTPM usage perspective, the only new feature that needs to be understood by a TPM user is the *pTPMCreated* attribute of objects and keys. To that effect, it is expected that a TSS exposes this new attribute, which is part of the creation template of an object or key. The new *pTPMCreated* attribute is part of the *TPMT_PUBLIC* structure. This means that the low-level API of a TSS such as the System API (SAPI) [27] does not need to change, although the marshalling code needs to recognise the new attribute. The TSS can detect if the *pTPMCreated* attribute is supported by a TPM by checking the version of the TPM specification supported, which is done by retrieving the TPM capabilities.

While all the new commands introduced by this proposal are meant to be used by a vTPM supporting the pTPM binding, a TSS can support the new commands for (i) ease of implementation of the vTPM and (ii) completeness.

8. Threat model

The solution proposed in this work strongly (i.e. at the hardware level) binds the vTPM state and management functions to the pTPM. Thus, it addresses the integrity and confidentiality threats to the virtual environment allowing for an hardware-grade assurance level.

Given the scenario described in Section 1.2, our reference threats target the virtual or hypervisor level, thus the malicious agents we consider have privileges of cloud service user, i.e. the user of the provided cloud service, or a malicious agent inadvertently running in the cloud service.

In principle, the vTPM implementations proposed in literature expose the virtual instances with the same trust and security functionalities as our proposal. However, due to the software nature of the implementation, in-memory and side-channel attacks are a significant menace that can affect these solutions, both at the virtual instance and at the hypervisor level.

Nowadays, many effective countermeasures (e.g. Stack cookies, exception handler validation, Data Execution Prevention, Address Space Layout Randomisation to name few of the most common) have been designed and implemented. Many methodologies, like secure coding, static and dynamic application analysis at the development stage make the exploitation of memory corruption bugs much harder. However, many tools and techniques to apply in-memory threats do exist (e.g. *use-after-free* and *double-free* based attacks to heap memory) still making direct memory manipulation a considerable attack set. Furthermore, some attacks – usually using side-channels – take advantage of weaknesses in the processor or memory modules themselves.

To protect from those attacks, the main approaches fall in three categories: (1) to apply access control protection to hypervisor, e.g. by means of AppArmor or SELinux policies, (2) to embed the software in a *Trusted Execution Environments* (TEE), so that they are protected against software attacks by hardware processor-specific safeguards or (3) to consistently bind the software vTPM to pTPM.

The later approach, which we propose in this solution, is more straightforward, since it allows a seamless extension of the Root of Trust to the whole software stack, from the boot to the virtual instance, not requiring a mixed and complex (and error-prone) approach to protect the whole stack of applications.

Furthermore, it appears more robust. Access control protection solutions are still vulnerable to many side-channel attacks (e.g. Specter, Meltdown [28], Rowhammer [29]), while even other hardware based approaches, like resources protected inside a TEE, suffers of effective attacks (e.g. downgrade attack [30]) to circumvent the security controls proposed. On the other hand, in our solution, we protect vTPM state and sensible data exploiting the hardware-level security functionality of the pTPM, protecting the vTPM state with the pTPM. Moreover, one of the main drawbacks of the TEE-based protection of a vTPM is the lack of flexibility, e.g. the problematic migration of the virtual instance to another node of the CSP infrastructure. Our solution offers a set of extensions to allow such flexibility while retaining the strong hardware binding.

The plethora of threats to confidentiality is not directly affected by our proposed extension. Meaning that the privacy protection given by the pTPM encryption and by the restriction imposed by the TPM specification to access pTPM protected keys still holds. Furthermore, the strong integrity assurance implied by our schema reduces the possibility to install malicious software in (virtual) environments, and thus reduces the chance to activate confidentiality threating agents (like a network sniffer or a keylogger in a virtual instance).

Service availability is of uttermost importance in a real production environment. It requires appropriate access control of vTPM to pTPM. In our scenario, many virtual instances can access the same pTPM (through the shared interface provided to the many vTPMs). This may result in simple-but-effective attacks like resource exhaustion (resulting in a DoS). This aspect can be addressed by complementary security controls. In particular, the CSP should provide either a selection of the vTPM access to the pTPM or mechanisms to prioritise and/or remove access from different vTPMs. Finally, our solution allows for a strong binding of vTPM to pTPM, but does not focus on availability of vTPM data blobs. Such protection of data storage, in scenarios where relevant, need to be addressed by different mechanisms (e.g. replication).

9. Discussion on our solution

In this section, we aim to highlight the possible drawbacks and critical aspects of the proposed solution. We start with analysing our solution applied to the use case from Section 1, where a CSP should provide their virtual appliances with hardware-bound vTPM virtual devices.

A cloud-based scenario counts several tens of VMs on a single physical machine and all of those could request a binding with the pTPM, when they are provisioned with a vTPM. The binding process involves the generation and wrapping of vTPM primary objects in each hierarchy of the pTPM. Depending on the type of object created, this can occupy the pTPM for a significant time: RSA key creation is measured in tens of seconds. This is acceptable for the virtual seeds parent keys, as they are created only once by the vTPM manager for all the vTPMs bound to the platform's pTPM.

After the initial setup, a virtual instance equipped with a vTPM could potentially create an unlimited number of *pTPMCreated* flagged objects. This behaviour in the interaction between a pTPM

and multiple vTPMs may introduce a risk for *Denial of Service* (DoS) on the host platform in case of vTPM malicious users having uncontrolled access to the physical device. This problem is particularly critical if vTPMs are exposed freely to containers in a serverless environment, as they can easily spin up and flood the pTPM interface. Because of this, there is a need for priority management and lockout in case of uncontrolled access to the pTPM by a specific vTPM instance. This could be achieved by leveraging the *TPM Access Broker and Resource Manager* (TABRM), as defined by TCG [31], which acts as an intermediate between the *Trusted Software Stack* (TSS) and the TPM device. Having smart TABRM could also limit other potential vulnerabilities due to the lack of access control. All vTPM contexts are isolated by default by the pTPM, which enforces the TPM enhanced authorisation mechanisms that leverage policy assertions [4]. In TPM 2.0, an assertion consists of a logical condition that is evaluated by the device to grant authorisation to a given object. Policies are composed of concatenating assertions via logical operators so that more advanced access control can be enforced. In particular, an assertion may require a set of PCRs to have specific values so that access to an object is authorised for use in a specific command. Even though this mechanism allows in principle to define granular policies for each vTPM and pTPM object, the assertions themselves must be specified carefully. Assertions can refer to the TPM state (e.g., a PCR value) as well as external secrets or inputs (e.g., a password given by an entity). In the first case, we need to find out a way to link the vTPM state to the pTPM.

In particular, when we create a hardware-protected object we could define policies according to the vTPM state (e.g. vTPM PCRs). Afterwards, the vTPM interface should map the command on the pTPM and define other policies for authorisation purposes at pTPM level. This means the final user of the vTPM is not aware of or involved in defining the policies at that level. Moreover, there is no link between vTPM state and pTPM. This results in a weakening of vTPM enhanced authorisation mechanism since the state is referring only to the software TPM. On the other hand, the vTPM user could use an external secret. This leads to the need of propagating the secret through the vTPM so that it can be used for enforcing the policy at pTPM level. The substantial advantage of this approach is that the vTPM user is the only one capable of accessing his pTPM-stored objects, avoiding attacks at the vTPM level. An initial workaround to this issue could be to restrict the use of policy assertions only to the ones that include external secrets. This leads to no further issues on both vTPM state and vTPM parameters retaining sides. In general, this new design enables new business models for the CSPs: access to pTPMs can now be part of their *Service Level Agreements* (SLAs). With our solutions, cloud customers can also verify that their vTPM is bound to a pTPM. Finally, if pTPMs are used to bind vTPMs and not identify an individual device, CSPs can equip their platforms with multiple pTPMs.

Other potential limitations could be encountered in the transient handle management by the vTPM manager. First of all, we aim to protect those handles from in-memory tampering attacks. In order to address this criticality we could use some in-memory protection technique as described in Section 5. In that case, we propose a trusted execution environment (e.g. SGX enclaves) to secure vTPM memory in a fully virtualised keys scenario. Now we focus on the protection of vTPM manager memory since it is in charge of the binding process and transient handle management. The availability as well as the consistency of data in that scenario is also another pivotal point. Since the handles managed by vTPM manager are the only way to replicate the status of the binding after a restart, it acts as a single point of failure. We should consider, in this regard, redundancy mechanisms to preserve those data, especially in a cloud scenario. Although our solution does not provide a way to protect against such kind of attacks, the vTPM manager can validate that a given handle references the correct virtual seeds parent key. Assuming the vTPM manager holds the certificate for an attestation key residing in the pTPM, the virtual seed parent key can be certified by the attestation key. Then the vTPM manager can validate that the handle references a pTPM object with the same public area as the certified key.

10. Future work

Future work includes the development of an initial *Proof of Concept* (PoC), which will leverage the TPM 2.0 simulator maintained by Microsoft[2] both for the extension of the pTPM interface and for the new vTPM internal mechanisms.

In addition to the implementation of a PoC, future research will target the vTPM-to-pTPM binding of integrity measurements, i.e. of vPCRs with the physical platform. This work will aim to implement the Deep Attestation use case, wherein remote attestation gives proof of the integrity of both the virtualised and physical platforms. Compared to existing approaches, we aim to root the vPCR values in the pTPM. The envisioned approach is to leverage pTPM NVRAM Extend indexes to record measurements of the virtualised instances. A deep quote operation may be implemented, leveraging the pTPM to sign the vPCR values with a hardware-rooted attestation key.

With this new design of vTPM architecture now formalised, Hewlett Packard Enterprise, which is a promoter of TCG, is interested in presenting the work to the TCG, particularly their Virtualized Platform [32] and TPM working groups. Feedback from more TPM experts will thus be gathered and our design may be included, even partially, in an upcoming standard proposal.

11. Conclusion

In this paper, we discuss a novel architecture to enable hardware-level security in vTPM instances running on top of a pTPM 2.0 equipped platform. The TPM virtualisation extension that we propose requires certain modifications at pTPM level so that the vTPM primary seed is managed as an internal object type by the device, and proper functions are added to generate vTPM primary keys and encrypt/decrypt their persistent state.

Compared to existing approaches, our solution offers different security levels depending on the required level of assurance, ranging from a fully hardware-rooted vTPM device to software management of vTPM keys and objects. Moreover, the vTPM primary objects (such as seeds) can be made duplicable, if intended, so that they can be migrated to a new pTPM host. Existing vTPM architectures are highly vulnerable to in-memory attacks, as the primary objects could be intercepted in the VM/host memory in clear text. In our approach, only vTPM child objects can be unsealed from the pTPM protected memory, offering hardware-level security for the most sensitive vTPM secrets.

CRediT authorship contribution statement

Marco De Benedictis: Conceptualization, Methodology, Investigation, Writing – original draft. **Ludovic Jacquin:** Conceptualization, Methodology, Writing – review & editing. **Ignazio Pedone:** Investigation, Validation, Writing – original draft. **Andrea Atzeni:** Investigation, Validation, Writing – original draft. **Antonio Lioy:** Conceptualization, Methodology.

[2] Microsoft TPM 2.0 Reference Implementation - https://github.com/microsoft/ms-tpm-20-ref

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgements

The work described in this paper has received funding by the European Union Horizon 2020 research and innovation programme, supported under Grant Agreement no. 883335 (PALANTIR project). This research activity has been conducted as a joint collaboration between Politecnico di Torino and Hewlett Packard Labs at the Security Lab of Hewlett Packard Enterprise in Bristol (UK).

References

[1] Trusted computing group website, 2021, URL: https://www.trustedcomputinggroup.org. (Accessed on 27 May 2021).
[2] Microsoft BitLocker overview, 2021, URL: https://docs.microsoft.com/windows/security/information-protection/bitlocker/bitlocker-overview. (Accessed on 27 May 2021).
[3] E. Brickell, J. Camenisch, L. Chen, Direct anonymous attestation, in: 11th ACM Conference on Computer and Communications Security, CCS'04, ACM, Washington (DC, USA), 2004, pp. 132–145, http://dx.doi.org/10.1145/1030083.1030103.
[4] Trusted platform module library part 1: Architecture family "2.0", 2021, URL: https://trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-1-Architecture-01.38.pdf. (Accessed on 27 May 2021).
[5] TCG EK credential profile for TPM family 2.0; level 0, 2021, URL: https://trustedcomputinggroup.org/wp-content/uploads/TCG_IWG_Credential_Profile_EK_V2.1_R13.pdf. (Accessed on 27 May 2021).
[6] Trusted platform module library part 3: Commands family "2.0", 2021, URL: https://trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-3-Commands-01.38.pdf. (Accessed on 27 May 2021).
[7] Trusted platform module library part 2: Structures family "2.0", 2021, URL: https://trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-2-Structures-01.38.pdf. (Accessed on 27 May 2021).
[8] R. Sailer, X. Zhang, T. Jaeger, L. van Doorn, Design and implementation of a TCG-based integrity measurement architecture, in: 13th USENIX Security Symposium, USENIX Association, San Diego (CA, USA), 2004, URL: https://www.usenix.org/legacy/publications/library/proceedings/sec04/tech/full_papers/sailer/sailer.pdf.
[9] Integrity Report Schema, Specification Version 2.0, Revision 5, Trusted Computing Group, 2021, URL: https://trustedcomputinggroup.org/wp-content/uploads/IWG_Integrity_Report_Schema_v2.0.r5.pdf. (Accessed on 27 May 2021).
[10] S. Berger, R. Cáceres, K.A. Goldman, R. Perez, R. Sailer, L. van Doorn, vTPM: Virtualizing the trusted platform module, in: 15th USENIX Security Symposium, USENIX Association, Vancouver (B.C., Canada), 2006, pp. 305–320, URL: https://www.usenix.org/legacy/events/sec06/tech/full_papers/berger/berger.pdf.
[11] Docker project website, 2021, URL: https://www.docker.com/. (Accessed on 27 May 2021).
[12] M. De Benedictis, A. Lioy, Integrity verification of Docker containers for a lightweight cloud environment, Future Gener. Comput. Syst. 97 (2019) 236–246, http://dx.doi.org/10.1016/j.future.2019.02.026.
[13] Y. Shi, B. Zhao, Z. Yu, H. Zhang, A security-improved scheme for virtual TPM based on KVM, Wuhan Univ. J. Nat. Sci. 20 (6) (2015) 505–511, http://dx.doi.org/10.1007/s11859-015-1126-5.
[14] S. Hosseinzadeh, S. Laurén, V. Leppänen, Security in container-based virtualization through vTPM, in: 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing, UCC, Shangai (China), 2016, pp. 214–219, http://dx.doi.org/10.1145/2996890.3009903.
[15] X. Wan, Z. Xiao, Y. Ren, Building trust into cloud computing using virtualization of TPM, in: 4th International Conference on Multimedia Information Networking and Security, Nanjing (China), 2012, pp. 59–63, http://dx.doi.org/10.1109/MINES.2012.82.
[16] P. England, J. Loeser, Para-virtualized TPM sharing, in: P. Lipp, A.R. Sadeghi, K.M. Koch (Eds.), Trusted Computing - Challenges and Applications, Springer Berlin Heidelberg, 2008, pp. 119–132, http://dx.doi.org/10.1007/978-3-540-68979-9_9.
[17] N. Schear, P.T. Cable II, T.M. Moyer, B. Richard, R. Rudd, Bootstrapping and maintaining trust in the cloud, in: 32nd Annual Conference on Computer Security Applications, ACSAC'16, ACM, Universal City (CA, USA), 2016, pp. 65–77, http://dx.doi.org/10.1145/2991079.2991104.
[18] A. Mosayyebzadeh, G. Ravago, A. Mohan, A. Raza, S. Tikale, N. Schear, T. Hudson, J. Hennessey, N. Ansari, K. Hogan, C. Munson, L. Rudolph, G. Cooperman, P. Desnoyers, O. Krieger, A secure cloud with minimal provider trust, in: 10th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'18, USENIX Association, Boston (MA, USA), 2018, URL: https://www.usenix.org/system/files/conference/hotcloud18/hotcloud18-paper-mosayyebzadeh.pdf.
[19] Intel software guard extensions project website, 2021, URL: https://software.intel.com/en-us/sgx. (Accessed on 27 May 2021).
[20] AMD secure encrypted virtualization project website, 2021, URL: https://developer.amd.com/sev/. (Accessed on 27 May 2021).
[21] ARM TrustZone technology website, 2021, URL: https://developer.arm.com/ip-products/security-ip/trustzone. (Accessed on 27 May 2021).
[22] J. Wang, F. Xiao, J. Huang, D. Zha, C. Fan, W. Hu, H. Zhang, A security-enhanced vTPM 2.0 for cloud computing, in: ICICS: International Conference on Information and Communications Security, Springer International Publishing, Beijing, (China), 2017, pp. 557–569, http://dx.doi.org/10.1007/978-3-319-89500-0_48.
[23] M. Zhou, S. Ruan, J. Liu, X. Chen, M. Yang, Q. Wang, vTPM-SM: An application scheme of SM2/SM3/SM4 algorithms based on trusted computing in cloud environment, in: 15th International Conference on Cloud Computing, CLOUD, IEEE, Barcelona (Spain), 2022, pp. 351–356, http://dx.doi.org/10.1109/CLOUD55607.2022.00058.
[24] D. Qian, S. Guo, L. Sun, Q. Hao, Y. Song, M. Wang, An integrity measurement scheme for containerized virtual network function, in: Journal of Physics: Conference Series, Vol. 2137, No. 1, IOP Publishing, 2021, 012029, http://dx.doi.org/10.1088/1742-6596/2137/1/012029.
[25] J. Wang, J. Wang, C. Fan, F. Yan, Y. Cheng, Y. Zhang, W. Zhang, M. Yang, H. Hu, SvTPM: SGX-based virtual trusted platform modules for cloud computing, IEEE Trans. Cloud Comput. (2023) http://dx.doi.org/10.1109/TCC.2023.3243891.
[26] D. Qian, S. Guo, L. Sun, H. Liu, Q. Hao, J. Zhang, Trusted virtual network function based on vTPM, in: 7th International Conference on Information Science and Control Engineering, ICISCE, IEEE, Changsha (China), 2020, pp. 1484–1488, http://dx.doi.org/10.1109/ICISCE50968.2020.00295.
[27] TCG TSS 2.0 system level API (SAPI) specification, 2023, URL: https://trustedcomputinggroup.org/resource/tcg-tss-2-0-system-level-api-sapi-specification. (Accessed on 28 July 2023).
[28] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, Meltdown: Reading kernel memory from user space, in: 27th USENIX Security Symposium (USENIX Security 18), USENIX Association, Baltimore (MD, USA), 2018, pp. 973–990, URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.
[29] K.S. Yim, The rowhammer attack injection methodology, in: IEEE Symposium on Reliable Distributed Systems, SRDS, Budapest (Hungary), 2016, pp. 1–10, http://dx.doi.org/10.1109/SRDS.2016.012.
[30] Y. Chen, Y. Zhang, Z. Wang, T. Wei, Downgrade attack on TrustZone, 2017, arXiv:1707.05082.
[31] TCG TSS 2.0 TAB and resource manager specification, 2021, URL: https://trustedcomputinggroup.org/wp-content/uploads/TSS_2p0_TAB_ResourceManager_v1p0_r18_04082019_pub.pdf. (Accessed on 27 May 2021).
[32] Virtualized Trusted Platform Architecture, Specification Version 1.0, Revision 0.26, Trusted Computing Group, 2021, URL: https://trustedcomputinggroup.org/wp-content/uploads/TCG_VPWG_Architecture_V1-0_R0-26_FINAL.pdf. (Accessed on 27 May 2021).

Marco De Benedictis holds a Ph.D. in computer and network security, as a member of the TORSEC research group at the Politecnico di Torino (Italy). His initial research activities were in the fields of electronic identity and digital signatures. His current interests are in security and trust of network and cloud infrastructures. De Benedictis received his M.Sc. in computer engineering from Politecnico di Torino.

Ludovic Jacquin is a principal investigator in the Research and Architecture team of the HPE Security Engineering group. His work mainly focuses on Trusted Computing technologies, methods, and protocols to enhance the security posture of platforms and infrastructures. He currently represents HPE in the Trusted Platform Module working group and actively contributes to the Infrastructure Work Group (Platform Certificate, TPM2.0 keys for Device Identity and Attestation) of the Trusted Computing Group. Ludovic holds a Master of Science degree in Applied Mathematics and Computer Science from ENSIMAG (Grenoble, France) and received his Ph.D. in Computer Science from Grenoble University (France) in 2013. He is also a $(ISC)^2$ Certified Information Systems Security Professional (CISSP) since 2019.

Ignazio Pedone received a M.Sc. degree and a Ph.D. in computer engineering from Politecnico di Torino. He is a former member of TORSEC Security Group at Politecnico di Torino and his research interests include the security of network infrastructures, quantum computing and networks, quantum and classical cryptography, and trusted/confidential computing.

Andrea Atzeni received the M.Sc. and Ph.D. degrees in computer engineering from the Politecnico di Torino. He is currently an Adjunct Professor at Politecnico di Torino. In last 20 years, he contributed to a number of large scale European research projects under the FP5, FP6, FP7, CIP, and Horizon 2020 programmes, addressing, among the others, the definition of security requirements in multi-platform systems, mobile security, modelisation of user expectation on security and privacy, security specification, risk analysis and threat modelling for complex cross-domain architectures, development of cross-domain usable security, digital and cloud forensics, development and integration of cross-border eIdentity, novel authentication mechanisms, malware analysis, and modelling.

Antonio Lioy received the M.Sc. degree in electronic engineering and the Ph.D. degree in computer engineering from the Politecnico di Torino. He currently leads the Cybersecurity Research Group TORSEC, Politecnico di Torino. He is also a full professor of cybersecurity. His research interests include electronic identity, PKI, trusted computing, and policy-based management of large IT systems.